Universidade do Minho
Escola de Engenharia

César de Jesus Pereira da Cunha Rodrigues

Foundations of Program Refinement by Calculation

Julho de 2008

César de Jesus Pereira da Cunha Rodrigues Foundations of Program Refinement by Calculation

UMinho | 2008

Universidade do Minho
Escola de Engenharia

César de Jesus Pereira da Cunha Rodrigues

Foundations of Program Refinement by Calculation

Tese de Doutoramento em Informática
Ramo de Fundamentos da Computação

Trabalho efectuado sob a orientação do
Professor Doutor José Nuno Fonseca de Oliveira

Julho de 2008

Aos meus pais, Cristina, Beatriz e Inês.

Sumário

Embora não seja prática generalizada, aceita-se hoje o valor da especificação formal de aplicações como ingrediente essencial ao desenvolvimento de *software* fiável. Isso pressupõe uma noção adicional — a de *refinamento* — capaz de sistematizar a derivação de implementações *correctas* a partir de modelos abstractos (ie. especificações).

No chamado *estilo construtivo* de desenvolvimento, faz-se refinamento passo-a-passo, provando que cada passo decorre do anterior por regras que garantem a correcção. Estas provas, que são vulgarmente feitas na lógica de predicados e teoria de conjuntos, têm, porém, problemas de escalabilidade: por um lado, não é prático provar factos envolvendo muitas variáveis e quantificações. Por outro, o nível relativamente pouco ágil em que decorrem as provas impede a sua progressão e pede ferramentas automáticas de prova.

Esta tese desenvolve uma técnica alternativa de refinamento baseada na chamada transformada-*pointfree*. A ideia é desenvolver um cálculo ágil capaz de calcular im plementações a partir das suas especificações por transformações algébricas simples. A transformada actua sempre que pretendemos raciocinar, mapeando expressões da lógica de predicados em expressões do cálculo relacional com implosão das quantificações e outras construções baseadas em variáveis.

Nesse sentido, esta tese aborda os fundamentos do refinamento de programas por cálculo, através de raciocínios ao nível do cálculo de relações binárias dito *pointfree*, nos seus dois níveis essenciais: dados e algoritmos.

Para esse efeito, desenvolvem-se e generalizam-se algumas construções do cálculo relacional, nomeadamente a transposição funcional, uma técnica que tem por objectivo converter relações em funções, de modo a exprimir a álgebra de relações através da álgebra de funções. É utilizada nesta dissertação como *leit-motiv*.

No sentido de potenciar ao máximo a pretendida algebrização do processo de cálculo de programas, a abordagem proposta capitaliza no conceito de conecção de Galois. Em particular, mostra-se como as principais leis de refinamento de dados podem ser vistas como esse tipo de conecção.

No plano do refinamento algorítmico, estuda-se a ordem padrão de refinamento ao nível *pointfree* e calcula-se a sua factorização em duas subordens com comportamentos opostos: redução de não-determinismo e aumento da definição. Essa factoriza ção torna a ordem original mais tratável matematicamente. Apresenta-se a sua teoria em estilo *pointfree*, que inclui uma prova simples do refinamento estrutural, para tipos paramétricos arbitrários.

Finalmente, mostramos que só precisamos de uma regra completa de refinamento relacional — para provar o refinamento coalgébrico — e utilizámo-la para testemunhar o refinamento por cálculo de relações de transição correspondentes a coalgebras.

Abstract

Design of trustworthy software calls for technologies which discuss software reliability formally, ie. by writing and reasoning about mathematical models of real-life objects and activities (vulg. *specifications*). Such technologies involve the additional notion of *refinement* (or *reification*), which means the systematic process of ensuring *correct implementations* for formal specifications.

In the well-known *constructive style* for software development, design is factored in several steps, each intermediate step being first proposed and then proved to follow from its antecedent. However, such an "invent-and-verify" style is often impractical due to the complexity of the mathematical reasoning involved in real-size software problems. Moreover, program reasoning is normally carried out in predicate/temporal logic and naïve set theory — notations which don't scale up to fully detailed models of complex problems.

This thesis is concerned with the foundations of an alternative technique for program refinement based on so-called *pointfree calculation*. The idea is to develop a *calculus* allowing for programs to be actually *calculated* from their specifications. Instead of doing proofs from first principles, this strategy leads to implementations which are *"correct by construction"*. Conventional refinement rules are transformed into simple, elegant equations dispensing with points and involving only binary relation combinators.

The pointfree binary relational calculus is therefore at the heart of the proposed refinement theory. This thesis adds to such a mathematical framework in two ways: on the one hand it shows how to apply it to data and algorithimc refinement problems. On the other hand, some constructions are proposed which prove useful not only in refinement but also in general. This includes generic functional transposition, a technique for converting relations into functions aimed at developing relational algebra *via* the algebra of functions. It is employed in this dissertation as a *leit motiv*.

Our proposed theory of data refinement draws heavily on the Galois connection approach to mathematical reasoning. This includes a simple way to calculate refinement invariants induced by the Galois connected laws.

Algorithmic refinement is addressed in the same way. The standard operation refinement ordering is given a pointfree treatment which includes a simple calculation of Groves' factorization and its direct application in structural refinement involving arbitrary parametric types.

Finally, coalgebraic refinement is done using an equivalent single complete rule for data refinement which is used to witness refinement by calculation of transition relations corresponding to coalgebras.

Agradecimentos

Agradeço ao Prof. José Nuno Oliveira por ter-me introduzido no mundo da investigação.

Aos colegas do DI agradeço todo o apoio que me deram. Em particular, gostaria de agradecer à Maria João, à Olga e à Cláudia.

Ao José Miguel agradeço o apoio que me deu.

Não podia deixar de agradecer aos meus pais, à minha irmã, e à Augusta, a força que me deram.

Contents

List of Figures

Chapter 1

Introduction

The Portuguese IT sector was hit in 2004 by a singular fact: soon after the MoEducation middle school teacher allocation system collapsed, officials said it would be better to revert to the traditional, manual process instead of trying to recover the existing, faulty software. When these facts reached the news headlines, the country as a whole asked obvious questions: what had happened to the investment in the IT sector? How good were our software professionals? What if similar problems arose elsewhere, eg. in banking software applications? How safe was it to let computers mechanize one's daily life?

Such a "national" problem was later to be solved by a small software house which claimed to use *formal methods* in their normal practice. For the first time ever in the country, issues such as program *correctness*, software (low) quality and so on reached the news headlines. For the first time ever, again, a software company decided to publish on their website a correctness proof of their proposed solution [5], an algorithm for teacher allocation.

Altogether, the incident drove the country's attention to the need for better trained software engineers. Skills as basic as the ability to think and reason in terms of abstract models and the effective use of mathematics and algorithmic science in normal, daily business practice are on demand.

This challenge of producing *correct* software, so well put forward by the facts reported above, is the main concern of this thesis.

1.1 On the specification-implementation dichotomy

Following common practice in other engineering disciplines, computing scientists have agreed that there must be at least two phases in the production of software: formal specification (modelling) and implementation. However, there are many ways in which these concepts are put into practice, ranging from completely informal (albeit systematic) strategies and guidelines, to fully formal reasoning techniques possbily involving the support of mechanical proof assistants.

Before explaining this specification-implementation dichotomy, let us stress that it demands the use of formal methods as a way of certifying the relationship between abstract models and efficient implementations [75]. In the specification phase, requirements should be mathematicaly expressed. By contrast, the implementation is a piece of code that should guide a computer to efficiently carry out the task intended by the specification.

It should be noted that, once the specification is written down, there is in general more than one way to instruct the target machine to accomplish "what" the specifier has designed. (Think, for instance, of the freedom to choose algorithms, data structures, programming languages, compilers, and so on.) Therefore, the relationship among specifications and implementations is one to many, that is, specifications are more abstract than implementations. In fact, when writing a specification, the software designer is aware that this is to be read, understood and reasoned about by human beings, who wish to think about the problem while abstracting as much as possible from machine-level details. By contrast, an implementation is intended to be run by a machine, as fast as possible. So, all programming tricks added to an implementation to improve its efficiency are irrelevant and meanningless details with respect to the original specification.

The "epistemological" gap between specification and implementation, which is far from beeing "smooth", can be approached by systematic introduction of detail. This process is studied by a discipline called *refinement* [66], a well established branch of computer programming engineering, based on formal methods.

There are two main approaches to program refinement: *invent & verify* and *correct by constuction*. In the former approach, an intermediate implementation is first invented and then proved correct in relation to the corresponding specification. In the latter method, the implementation is actualy calculated from the specification, by use of algebraic reasoning. Altogether, the theory and practice behind these two approaches is the main body of what is currently understood as the use of so-called *formal methods* in software design.

The notion of *formal correctness* is therefore central to any reliable refinement discipline. Examples of notations and methodologies promoting *correctness* by formal reasoning are Z [142], B [2] and VDM [76, 54], the latter being among the first to promote a comprehensive methodology for formal specification and development of programs. Quoting Cliff Jones [77],

> *"What is required is a coherent notation and a set of 'proof obligations'. The so-called 'Vienna Development Method' (VDM) was one of the earliest attempts to create such coherence".*

1.2 Balzer's software life-cycle

As we have seen above, formal methods are based on formal specifications. A formal specification (in which a mathematical text — the model — is written prescribing "what" the intended software system should do) is linked to (one or more) implementations (in which machine code is produced instructing the hardware about "how" to do it) [75]. The intended coherence between these two phases is achieved by means of

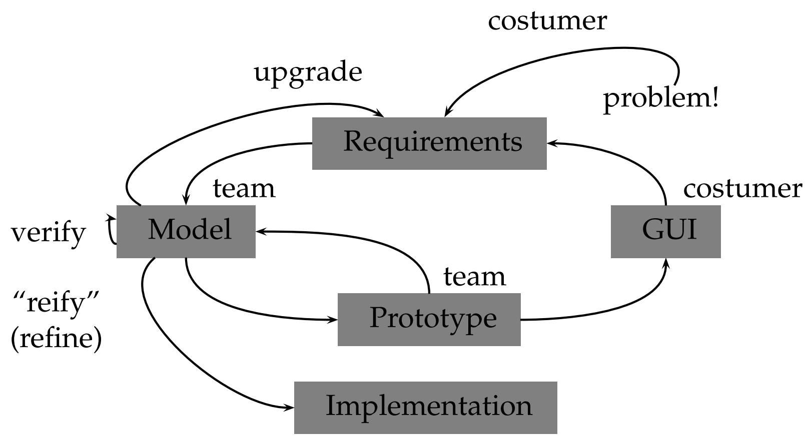

Figure 1.1: Diagram of Balzer's software life-cycle [17] (adapted)

a justification, a mathematical document saying "why" the implementation meets the abstract model [75].

This approach is the basis of the so-called Balzer's life-cycle proposal [17] for formal software development illustrated in the diagram of figure 1.1. In essence, we have the following steps:

- A costumer poses a problem which is understood textually via informal requirements (step 1).
- The development team builds a mathematical model of the requirements, that is, the formal specification. This captures the formal understanding of the problem by the team (step 2).
- The team checks the behaviour of the model by animating it as a rapid prototype written in a high level declarative language. Functional behaviour can be checked reactively, while logical properties and conditions require suitable test suites. In the case of reactive prototyping, the runnable emulation of the model can be wrapped by a graphical user interface (GUI) and exercised directly by the costumer, who will thus validate the original requirements (step 3).
- The team upgrades the model (specification) wherever inconsistencies or misinterpretations have been found; ocasionally, changes to the original requirements are made which the costumer finds appropriate for best tuning of the model to the problem (step 4).
- Once the model is stable, formal verification can take place (step 5) where consistency proof obligations are discharged (invariant property checking, for instance).

In the software life cycle, we give particular attention to the reification phase (step 5), where an implementation is materialized into a lower level program. Let us see what the Wikipedia tells about refinement:

> *In formal methods,* refinement *is the verifiable transformation of an abstract (high-level) formal specification into a concrete (low-level) executable program. The term* reification *is also sometimes used (coined by Cliff Jones).*
> Retrenchment *is an alternative technique when formal refinement is not possible.*

Note that reify m means turn m into something "real". However, instead of *reify* (= "real+fy") we tend to use the more traditional verb *refine*.

In the following section we'll see how one

- refines a given model
- checks the *validity* of the outcome of such a refinement phase.

1.3 On the Need for Software Refinement Calculi

The problem of program verification appears in the larger context of so called program (or code) *validation*, that is, of the quality control of software when viewed as an industrial product.

The first form of quality control of a program ever known is called test (or *debug*). It consists in correcting a program after successive runs, in a process of trial and error. Its drawbacks are well known, cf. [110, 75]:

- it is clearly a pre-scientific methodology;
- it is ineffective, as E.W. Dijkstra noted:

  > *Program testing can be used to show the presence of bugs, never to show their absence*

  So,

  > debug reveals errors $\Rightarrow$ program has errors ($p \Rightarrow q$)
  > debug does'nt reveals errors $\not\Rightarrow$ program does'nt have errors ($\neg p \not\Rightarrow \neg q$)

- it is a method of restricted application (not all test cases can be generated);
- it is an expensive method since trial and error demands extra time and more financial charges (programmers' wages, etc);
- it implies excessive use of computational resources, adding to the financial costs of development because of the need for additional computers and disk space to store test suits;
- it is an irresponsable method as the human incapability of testing until the end will not blame programmers and will dismiss professional responsability, which is contrary to the spirit of the modern technology;

- it produces insufficient results, since experience shows that many errors detected in the debugging phase were originated earlier in the development process; to remove such errors at the end (in place of their timely detection) is obviously more difficult and demanding — e.g. to recover bad choice of a data structure may force the substitution of all algorithms associated with it;

- finally, it is a method which makes the maintenance process harder thus increasing cost of the overall software production.

An alternative way to control the quality of programs — the so called *verification* approach — is different since it tries to verify the correctness of a program without ever runing it. This software quality control appeared in the 1960s after research activities which showed the possibility of "calculating" the mathematical meaning of a computer program [55, 65].

In this way, the quality control process is transformed in a mathematical exercice. But for real programs, with thousands of lines of code, the calculation of their meaning and associated correctness argument with respect to a given specfication, is a very hard task. Thus the "historical failure" of the method, today called static verification or *a posteriori* verification.

The term *a posteriori* retains the fact that the correction argument only starts once the program is finished. Common sense suggests that correction arguments should be mind sized. So, for real programs, this is only possible if the argument begins *before* the synthesis of the final program to develop. The idea is then to reduce the complexity of the correctness argument by structuring it in sub-arguments, in the same way a mathematician decomposes an elaborate theorem into auxiliary theorems (lemmas), which should be "mind sized", and are proven in isolation [110].

In this new style of formal development, referred to as *constructive specification* or 'invent and verify' [75], a design is factored into as many "mind-sized" design *steps* as required. Every intermediate design is first **proposed** and then **proved** to follow from its antecedent. Although it ameliorates primitive static verification a lot, this style (known as "invent and verify") is quite often ineffective for it assumes that the software engineer has sufficient intuition to guess efficient implementations, which is unlikely in many cases. The complexity of the mathematical reasoning demanded in proving intermediate steps of the development of software solutions for real life problems is still high.

The questions arise: since the whole process is based, inductively, on conjecture and invention, what's the quality of the invented implementations? Isn't it possible to derive (to deduce) implementations from ther specifications?

In face of the limitations of this method, we propose a calculation style:

- **Idea**: develop a *calculus* allowing *programs* to be actually calculated from their specifications;

- **Style**: every intermediate design is drawn from its antecedent by transformation according to some underlying calculus;

- **Proof discharge**: one is lead to a *"correct by construction"* method (no proofs from first principles).

This style calls for

- a compact *symbolic* notation for code and data structures, for instance $x + y$ instead of `record case ...of ...`x` ...`y` end`, etc.
- program calculi involving mathematical *laws* as simple and familiar as eg. $x + y \leq z \equiv x \leq z - y$, which are well understood since school algebra.

However, what can the meaning of $x \leq y$ be where x and y are programming concepts? From Wirth's "formula" [141]

Algorithms + Data Structures = Programs

we infer

Program refinement = Data refinement + Algorithmic refinement

subject to the following remarks:

- in general, $x \leq y$ will be read as "y is a *refinement* of x" (whatever x, y are);
- in **algorithmic** refinement, $x \leq y$ means program y is more defined and more deterministic than x;
- in **data** refinement, $x \leq y$ means that every inhabitant of datatype x can be (faithfully) represented by some inhabitant of datatype y;
- *data* should be dealt with first.

Both data and algorithm calculations should be structural in the sense that the components of an expression be calculated in isolation (i.e. pre-existing refinement results may be reused). The reduction of the proof onus is accomplished doing structural calculation instead of proofs from first principles. This is the very idea of a *calculus*, as the past has witnessed in other contexts (cf. the differential and integral calculi, linear algebra, etc).

We are thus lead to the following diagram picturing the evolution of the software quality control techniques toward automation:

$$\text{Code Validation} \left\{ \begin{array}{l} \text{Debug} \\ \text{Verification} \left\{ \begin{array}{l} \text{Static } (\textit{a posteriori}) \\ \text{Constructive } (\textit{invent \& verify}) \end{array} \right. \\ \textbf{Calculation} \text{ / automation} \end{array} \right.$$

The challenge of turning the art of computer programming into a calculational discipline is nicely captured by the following advice concerning software calculi, by the Algebra of Programming Research Group at Oxford [29]:

a calculus is worthless if it is not sound, and useless if it is not easy to work with.

1.4 State of the Art

Software calculi have been the subject of intensive research in the last twenty years. Among the most popular algorithmic refinement calculi we find Morgan's calculus [94] and Back's calculus [7]. These are generalizatons of Dijkstra's calculus [52] which were developed in the late 1980's. Earlier publications concerning the Morgan's calculus are reprinted in [96] and include [93]. Ralf Back's calculus is the subject of many publications, among which [8, 6, 139, 9]. The so-called Morris's calculus [97, 98] is similar to the above.

More recently, in the field of functional programming, an algebraic approach to programming [29] has emerged that builds on earlier work by Backus [16] on the *algebra of programming*. A *pointfree* algebraic style is put forward which proves effective in calculating with functions and relations. (This calculus, which meanwhile became known as the Bird-Meertens formalism [30, 86, 31, 11], provides the main inspiration for the work presented in this thesis.)

Also in the area of functional programming one finds the fold/unfold calculus [36] and the program transformation approach [44, 45, 46]. The CIP [25, 90, 120] program transformation calculus has lead to the CIP transformation framework [24] over an wide spectrum language [26].

As far as data calculation is concerned, the so-called *set-specification* calculus SETS [104, 105, 106, 108, 116] emerged in the context of [66] and VDM [75, 76]. The foundations of this calculus were studied in [79].

Relational data refinement [50] (see also [64, 63]) is the theory of implementation of abstract data types by concrete ones, by use of proof rules based on relations called simulations.

Algorithmic refinement may be seen as a particular case of relational refinement. Consider the VDM and Z tradition on refinement, and references [35] and [61], the latter in the context of the Z schema calculus.

The literature on refinement includes comparisions among data (relational) refinement [50] and process refinement [62] approaches. The most significant examples (see e.g. [34, 33]) arise in the context of the relationship between refinement in Z [135] and in CSP[67]. Reference [53] compares refinement for failure (typical of CSP) and bissimulation (typical of CCS [88]) semantics. Bisimulation is also an important feature of the universal coalgebra theory of systems [132, 4].

In the context of coalgebra, [18] studies the semantics of software components and processes. Processes are given a final coalgebraic semantics [19, 20].

The following references are textbooks on formal software calculi:

- Reference [120] presents the CIP wide spectrum language and *transformations*, between the various levels of the language. Specifications, corresponding to the more abstract level of the language, are algebraic and axiomatic.

- Reference [93] presents a formalization of Dijkstra's methodology, which involves an imperative algorithmic language (Dijkstra's language) with non-executable extensions. Specifications are mathematical statements and the implementations are programs in Dijkstra's language.

- Reference [29] presents a categorical formalization of the binary relation calculus, where specifications are relations and implementations are functions.

- Reference [7] addresses the refinement calculus, an alternative formalization of Dijkstra's ideas.

1.5 Structure of the Dissertation

Chapter 2 presents the pointfree binary relational calculus and the pointfree transform, a device which will be used throughout the dissertation to convert first order logic formulas to pointfree binary relational calculus expressions.

Chapter 3 develops the concept of transposition, which will be used in the remaining chapters.

Chapter 4 is devoted to data refinement, ie. the refinement of the carrier of an algebra or else the carrier of a coalgebra.

Chapter 5 is devoted to algorithmic refinement, ie. the refinement of the operations of an algebra. A programming calculus is developed which has relations as specifications and functions as implementations.

Chapter 6 is devoted to coalgebraic refinement, ie. the refinement of the dynamics of a coalgebra. Coalgebraic refinement is shown to reduce to relational refinement of the respective binary transition relations.

The last chapter presents conclusions and directions of future work.

1.6 Sources of chapters

Chapter 3 is an extended version of paper [117]. Similarly, Chapter 5 extends paper [118].

Part I

Mathematical Background

Chapter 2

On the Binary Relation Algebra

2.1 Introduction

The functional programming style emerged in the 1970s as an alternative to programming with actions — also known as imperative programming – amenable to reasoning and calculation, cf. Backus' Turing Award paper [16] and the well known *program transformation* school [36, 45, 90]. But soon it became apparent that this functional style integrates a more complete approach where specifications are relations and implementations are functions [29].

With relations one is able to model partiality and non-determinism. Relations also include predicates, invariants, loop invariants, sets, orders and other relations useful to avoid over-specification. The most important aspect of this shift from functions to relations is the powerful calculus of the latter, which can assist in refinement. Such a 'calculational style' was introduced in the 1980s [68]. This calculus, also known as the binary relation algebra, benefits from the point-free transform, in the main subject of this chapter.

The study of binary relations is, however, much older. It begun in the nineteenth century, cf. [47, 121, 134] and was recovered by Tarski [136] in the 1940s. References [124, 83] review the origins of the calculus of binary relations. References [91, 92] generalize from binary to multiary relations, and apply the resulting approach to graph and pointer algorithms. Another generalization was to drop converse: [138] and [27]. The resulting theory was applied to the development of a process calculus.

2.2 Overview of the relational calculus

Relations. Let $B \xleftarrow{R} A$ denote a binary relation on datatypes A (source) and B (target). We write bRa to mean that pair $\langle b, a\rangle$ is in R. The underlying partial order on relations will be written $R \subseteq S$, meaning that S is either more defined or less deterministic than R, that is, $R \subseteq S \quad \equiv \quad bRa \Rightarrow bSa$ for all a, b. Expression $R \cup S$

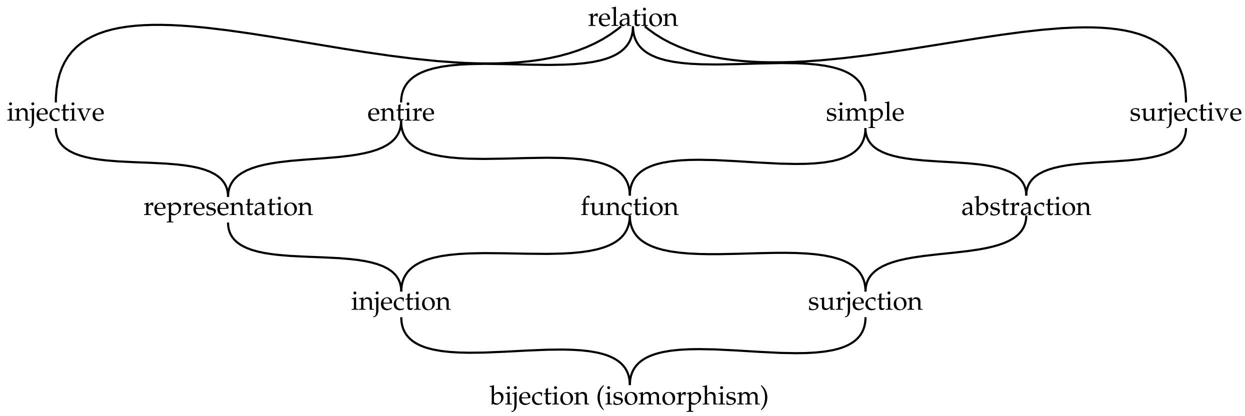

Figure 2.1: Binary relation taxonomy

denotes the union of two relations and $\top$ denotes the largest relation of its type. Its dual is $\perp$, the smallest such relation. Equality on relations can be established by $\subseteq$-antisymmetry: $R = S \equiv R \subseteq S \ \wedge \ S \subseteq R$.

Relations can be combined by three basic operators: composition ($R \cdot S$), converse (R°) and meet ($R \cap S$). R° is the relation such that $a(R^\circ)b$ iff bRa holds. Meet corresponds to set-theoretical intersection and composition is defined in the usual way: $b(R \cdot S)c$ holds wherever there exists some mediating $a \in A$ such that $bRa \ \wedge \ aSc$. Everywhere $T = R \cdot S$ holds, the replacement of T by $R \cdot S$ will be referred to as a "factorization" and that of $R \cdot S$ by T as "fusion". Every relation $B \xleftarrow{R} A$ admits two trivial factorizations, $R = R \cdot id_A$ and $R = id_B \cdot R$ where, for every X, id_X is the identity relation mapping every element of X onto itself.

Coreflexives. Some standard terminology arises from the id relation: a (endo)relation

$$A \xleftarrow{R} A$$

(often called an *order*) will be referred to as *reflexive* iff $id_A \subseteq R$ holds and as *coreflexive* iff $R \subseteq id_A$ holds. As a rule, subscripts are dropped wherever types are implicit or easy to infer.

Coreflexive relations are fragments of the identity relation which can be used to model predicates or sets. The meaning of a *predicate* p is the coreflexive $[\![p]\!]$ such that $b[\![p]\!]a \ \equiv \ (b = a) \ \wedge \ (p \ a)$, that is, the relation that maps every a which satisfies p (and only such a) onto itself. The meaning of a *set* $S \subseteq A$ is $[\![\lambda a.a \in S]\!]$, that is, $b[\![S]\!]a \ \equiv \ (b = a) \ \wedge \ a \in S$. Wherever clear from the context, we will omit the $[\![\ ]\!]$ brackets.

Orders. Preorders are reflexive, transitive relations, where R is transitive iff $R \cdot R \subseteq R$ holds. Partial orders are anti-symmetric preorders, where R is anti-symmetric wherever $R \cap R^\circ \ \subseteq id$ holds. A preorder R is an *equivalence* if it is symmetric, that is, if $R = R^\circ$.

Taxonomy. Converse is of paramount importance in establishing a wider taxonomy of binary relations. Let us first define two derived operators, the so-called *kernel* of a relation

$$\mathit{ker}\, R \stackrel{\text{def}}{=} R^\circ \cdot R \tag{2.1}$$

and its *image* (dual concept)

$$\mathit{img}\, R \stackrel{\text{def}}{=} \mathit{ker}\, (R^\circ) \tag{2.2}$$

An alternative to (2.2) is to define $\mathit{img}\, R = R \cdot R^\circ$, since converse commutes with composition,

$$(R \cdot S)^\circ = S^\circ \cdot R^\circ \tag{2.3}$$

and is involutive, that is,

$$(R^\circ)^\circ = R \tag{2.4}$$

Kernel and image lead to the following terminology: a relation R is said to be *entire* (or total) iff its kernel is reflexive; or *simple* (or functional) iff its image is coreflexive. Dually, R is *surjective* iff R° is entire, and R is *injective* iff R° is simple. This terminology is recorded in the following summary table:

	Reflexive	*Coreflexive*
$\mathit{ker}\, R$	entire R	injective R
$\mathit{img}\, R$	surjective R	simple R

(2.5)

Functions. A relation is a *function* iff it is both simple and entire. Functions will be denoted by lowercase letters (f, g, etc.) and are such that $b f a$ means $b = f\ a$. Function converses enjoy a number of properties of which the following is singled out because of its rôle in pointwise-pointfree conversion [10] :

$$b(f^\circ \cdot R \cdot g)a \quad \equiv \quad (f\ b)R(g\ a) \tag{2.6}$$

The overall taxonomy of binary relations is pictured in Fig. 2.1 where, further to the standard classification, *representations* and *abstractions* are added. These are classes of relations useful in data-refinement [106]. Because of $\subseteq$-antisymmetry, $\mathit{img}\, S = id$ wherever S is an *abstraction* and $\mathit{ker}\, R = id$ wherever R is a *representation*. This ensures that "no confusion" arises in a representation and that all abstract data are reachable by an abstraction ("no junk").

Isomorphisms are functions, abstractions and representations at the same time. A particular isomorphism is id, which also is the smallest equivalence relation on a particular data domain. So, $b\ id\ a$ means the same as $b = a$.

Functions and relations. The interplay between functions and relations is a rich part of the binary relation calculus. This arises when one relates the arguments and results of pairs of functions f and g in, essentially, two ways:

$$f \cdot S \quad \subseteq \quad R \cdot g \tag{2.7}$$
$$f^{\circ} \cdot S \quad = \quad R \cdot g \tag{2.8}$$

As we shall see shortly, (2.7) is equivalent to $S \subseteq f^{\circ} \cdot R \cdot g$ which, by (2.6), means that f and g produce R-related outputs $f\ b$ and $g\ a$ provided their inputs are S-related (bSa). This situation is so frequent that one says that, everywhere f and g are such that (2.7) holds, f is $(R \leftarrow S)$*-related* to g:

$$f(R \leftarrow S)g \quad \equiv \quad f \cdot S \subseteq R \cdot g \qquad \text{cf. diagram} \qquad \begin{tikzcd} B \arrow[d, "f"'] \arrow[dr, phantom, "\subseteq"] & A \arrow[l, "S"'] \arrow[d, "g"] \\ C & D \arrow[l, "R"] \end{tikzcd} \tag{2.9}$$

For instance, for partial orders $R, S := \leq, \sqsubseteq$, fact $f(\leq \leftarrow \sqsubseteq)f$ means that f is monotone. For $R, S := \leq, id$, fact $f(\leq \leftarrow id)g$ means

$$f \mathrel{\dot{\leq}} g \quad \equiv \quad f \quad \subseteq \quad \leq \cdot\, g \tag{2.10}$$

that is, f and g are such that $f\ b \leq g\ b$ for all b. Therefore, $\dot{\leq}$ is the lifting of pointwise ordering $\leq$ to the functional level. In general, relation $R \leftarrow S$ will be referred to as "Reynolds arrow combinator" (see section 3.5), which is extensively studied in [10].

Wherever $f(R \leftarrow S)f$ holds, we will write

$$R \xleftarrow{f} S \tag{2.11}$$

instead of $f(R \leftarrow S)f$ meaning that "f is of type $R \leftarrow S$". This will be of special interest where R and S are coreflexives expressing properties of datatypes. So, writing $\leq \xleftarrow{f} \sqsubseteq$ is an alternative way of telling that f is monotone.

Concerning the other way to combine relations with functions, equality (2.8) becomes interesting wherever R and S are preorders,

$$f^{\circ} \cdot \sqsubseteq \quad = \quad \leq \cdot\, g \qquad \text{cf. diagram:} \qquad \begin{tikzcd} B \arrow[loop left, "\leq"] \arrow[r, bend left, "f"] & C \arrow[loop right, "\sqsubseteq"] \arrow[l, bend left, "g"] \end{tikzcd} \tag{2.12}$$

in which case f, g are always monotone and said to be *Galois connected*. Function f (resp. g) is referred to as the *lower* (resp. *upper*) adjoint of the connection. By introducing variables in both sides of (2.12) via (2.6) we obtain, for all b and c

$$(f\ b) \sqsubseteq c \quad \equiv \quad b \leq (g\ c) \tag{2.13}$$

Note that (2.12) boils down to $f^{\circ} = g$ (ie. $f = g^{\circ}$) wherever $\leq$ and $\sqsubseteq$ are id, in which case f and g are isomorphisms, that is, f° is also a function and

$$f\ b = c \quad \equiv \quad b = f^{\circ} c \tag{2.14}$$

holds.

For further details on the rich theory of Galois connections and examples of application see [1, 10]. Galois connections in which the two preorders are relation inclusion ($\leq, \sqsubseteq := \subseteq, \subseteq$) are particularly interesting because the two adjoints are relational combinators and the connection itself is their universal property. The following table lists connections which are relevant for this dissertation:

Relational Operators as Galois Connections			
$(f\ X) \subseteq Y \equiv X \subseteq (g\ Y)$			
Description	$f = g^\flat$	$g = f^\sharp$	**Obs.**
converse	$(\_)^\circ$	$(\_)^\circ$	
shunting rule	$(h\cdot)$	$(h^\circ\cdot)$	
“converse” shunting rule	$(\cdot h^\circ)$	$(\cdot h)$	
left-division	$(R\cdot)$	$(R \setminus\ )$	R under
right-division	$(\cdot R)$	$(\ / R)$	over R
range	ρ	$(\cdot \top)$	
domain	δ	$(\top \cdot)$	
implication	$(R \cap\ )$	$(R \Rightarrow\ )$	
difference	$(\_ - R)$	$(R \cup\ )$	

(2.15)

All f and g are monotonic by definition, as Galois adjoints. Moreover, the fs commute with join and the gs with meet. Thus we obtain monotonicity and distribution for free, whose proof as law 3.2 in [61] is unnecessary. It should be mentioned that some rules in (2.15) appear in the literature under different guises and usually not identified as Galois connections. For instance, the *shunting* rule is called *cancellation law* in [142].

The connection associated with the *domain* operator will be particularly useful later on, whereby we infer that it is monotonic and commutes with join

$$\delta\,(R \cup S) \quad = \quad (\delta\,R) \cup (\delta\,S) \tag{2.16}$$

(as all lower-adjoints do) and can be switched to so-called *conditions* [69]

$$\delta\,R \subseteq \delta\,S \quad \equiv \quad ! \cdot R \subseteq\, ! \cdot S \tag{2.17}$$

wherever required, since $\top = \mathit{ker}\,!$.

Left-division is another relational combinator relevant for this dissertation, from whose connection in (2.15) not only the following pointwise definition can be inferred [12],

$$b\,(R \setminus Y)\,a \quad \equiv \quad \langle \forall\, c \;:\; c\,R\,b :\; c\,Y\,a \rangle \tag{2.18}$$

but also the following properties which will be useful in the sequel, for Φ coreflexive:

$$(R \cup T) \setminus S \quad = \quad (R \setminus S) \cap (T \setminus S) \tag{2.19}$$

$$((R \cdot \Phi) \setminus S) \cap \Phi \quad = \quad (R \setminus S) \cap \Phi \tag{2.20}$$

From the two Galois connections in (2.15) called *shunting rules* one infers the very useful fact that equating functions is the same as comparing them in either way:

$$f = g \quad \equiv \quad f \subseteq g \quad \equiv \quad g \subseteq f \tag{2.21}$$

Relators. A *relator* [14] is a concept which extends *functors* to relations: $\mathsf{F}\,A$ describes a parametric type while $\mathsf{F}\,R$ is a relation from $\mathsf{F}\,A$ to $\mathsf{F}\,B$ provided R is a relation from A to B. Relators are monotone and commute with composition, converse and the identity.

The most simple relators are the *identity* relator Id, which is such that $\mathsf{Id}\ A = A$ and $\mathsf{Id}\ R = R$, and the *constant* relator K (for a particular concrete data type K) which is such that $\mathsf{K}\ A = K$ and $\mathsf{K}\ R = id_K$.

Relators can also be multi-parametric. Two well-known examples of binary relators are product and sum,

$$R \times S \quad = \quad \langle R \cdot \pi_1, S \cdot \pi_2 \rangle \tag{2.22}$$

$$R + S \quad = \quad [i_1 \cdot R, i_2 \cdot S] \tag{2.23}$$

where π_1, π_2 denote the projection functions of a Cartesian product, i_1, i_2 denote the injection functions of a disjoint union, and the *split/either* relational combinators are defined by

$$\langle R, S \rangle \quad = \quad \pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S \tag{2.24}$$

$$[R, S] \quad = \quad (R \cdot i_1^\circ) \cup (S \cdot i_2^\circ) \tag{2.25}$$

By putting these four kinds of relator (product, sum, identity and constant) together with fixpoint definition one is able to specify a large class of parametric structures — called *polynomial* — such as those implementable in Haskell. For instance, the $Maybe$ datatype is an implementation of polynomial relator $\mathsf{F} = \mathsf{Id}+1$ (ie. $\mathsf{F}\ A = A+1$), where 1 denotes the *singleton* datatype, written $()$ in Haskell [78].

Membership. Recall the notion of membership from set theory. Wherever we write $a \in x$, where x is a set, we mean a relation of type $A \xleftarrow{\in} \mathcal{P}(A)$, where $\mathcal{P}(A)$ denotes the set of all subsets of A.

Sentence $a \in x$ (meaning that "a belongs to x" or "a occurs in x") can be generalized to x's other than sets. For instance, one may check whether a particular integer occurs in one or more leaves of a binary tree, or of any other *collective* or *container* type F.

Such a generic membership relation will have type $A \xleftarrow{\in} \mathsf{F}\,A$, where F is a type *parametric on A*. Technically, the parametricity of F is captured by regarding it as a *relator*.

There is more than one way to generalize $A \xleftarrow{\in} \mathcal{P}(A)$ to relators other than the powerset. (For a thorough presentation of the subject see chapter 4 of [69].) For the purpose of this dissertation it will be enough to say that $A \xleftarrow{\in_{\mathsf{F}}} \mathsf{F}\,A$, if it exists, is a *lax natural transformation* [29], that is,

$$\in_{\mathsf{F}} \cdot \mathsf{F}\,R \quad \subseteq \quad R \cdot \in_{\mathsf{F}} \tag{2.26}$$

holds. Moreover, relators involving $+, \times, \mathsf{Id}$ and constants have membership defined inductively as follows:

$$\in_{\mathsf{K}} \quad \stackrel{\mathrm{def}}{=} \quad \bot \tag{2.27}$$
$$\in_{\mathsf{Id}} \quad \stackrel{\mathrm{def}}{=} \quad id \tag{2.28}$$
$$\in_{\mathsf{F}\times\mathsf{G}} \quad \stackrel{\mathrm{def}}{=} \quad (\in_{\mathsf{F}} \cdot \pi_1) \cup (\in_{\mathsf{G}} \cdot \pi_2) \tag{2.29}$$
$$\in_{\mathsf{F}+\mathsf{G}} \quad \stackrel{\mathrm{def}}{=} \quad [\in_{\mathsf{F}}, \in_{\mathsf{G}}] \tag{2.30}$$

To complete the definition for so called *polynomial relators* we only need to give the membership rule for relator composition:

$$\in_{\mathsf{F}\cdot\mathsf{G}} \quad \stackrel{\mathrm{def}}{=} \quad \in_{\mathsf{F}} \cdot \in_{\mathsf{G}} \tag{2.31}$$

2.3 On the Pointfree Transform

The main purpose of formal modelling is to identify properties of real-world situations which, once expressed by mathematical formulæ, become abstract models which can be queried and reasoned about. This often raises a kind of *notation* conflict between *descriptiveness* (ie., adequacy to describe domain-specific objects and properties, inc. diagrams or other graphical objects) and *compactness* (as required by algebraic reasoning and solution calculation).

Classical *pointwise* notation in logic involves operators as well as variable symbols, logical connectives, quantifiers, etc. in a way which is hard to scale-up to complex models. This is not, however, the first time this kind of notational conflict arises in mathematics. Elsewhere in physics and engineering, people have learned to overcome it by changing the "mathematical space", for instance by moving (temporarily) from the t-space (t for time) to the s-space in the *Laplace transformation* (fig. 2.2). Quoting [81], p.242 [1]:

> *The Laplace transformation is a method for solving differential equations (...) The process of solution consists of three main steps:*
>
> **1st step.** *The given "hard" problem is transformed into a "simple" equation (*subsidiary equation*).*
>
> **2nd step.** *The subsidiary equation is solved by **purely algebraic** manipulations.*
>
> **3rd step.** *The solution of the subsidiary equation is transformed back to obtain the solution of the given problem.*
>
> *In this way the Laplace transformation reduces the problem of solving a differential equation to an **algebraic problem**.*

The *pointfree (PF) transform* adopted in this dissertation is at the heels of this old reasoning technique. Standard set-theory-formulated refinement concepts — such as eg. (5.1) — are regarded as "hard" problems to be transformed into "simple", *subsidiary equations* dispensing with points and involving only binary relation concepts. As in the Laplace transformation, these are solved by *purely algebraic* manipulations and the outcome is mapped back to the original (descriptive) mathematical space wherever required.

Note the advantages of this two-tiered approach: intuitive, domain-specific descriptive formulæ are used wherever the model is to be "felt" by people. Such formulæ are transformed into a more *elegant*, simple and compact — but also more cryptic — algebraic notation whose single purpose is easy manipulation.

The pointfree transform is a calculus that gives the pointfree version of a particular pointwise relational expression,

$$
\begin{array}{c|c}
\phi & PF\ \phi \\
\hline\hline
\langle \exists\, a\ ::\ b\,R\,a\ \wedge\ a\,S\,c \rangle & b(R \cdot S)c \\
\langle \forall\, a, b\ ::\ b\,R\,a \Rightarrow b\,S\,a \rangle & R \subseteq S \\
\langle \forall\, a\ ::\ a\,R\,a \rangle & id \subseteq R \\
\langle \forall\, x\ ::\ x\,R\,b \Rightarrow x\,S\,a \rangle & b(R \setminus S)a \\
\langle \forall\, c\ ::\ b\,R\,c \Rightarrow a\,S\,c \rangle & a(S \,/\, R)b \\
b\,R\,a\ \wedge\ c\,S\,a & (b,c)\langle R, S \rangle a \\
b\,R\,a\ \wedge\ d\,S\,c & (b,d)(R \times S)(a,c) \\
b\,R\,a\ \wedge\ b\,S\,a & b\,(R \cap S)\,a \\
b\,R\,a \vee b\,S\,a & b\,(R \cup S)\,a \\
(f\ b)\ R\ (g\ a) & b(f^{\circ} \cdot R \cdot g)a \\
\textsc{True} & b\ \top\ a \\
\textsc{False} & b\ \bot\ a
\end{array}
\tag{2.32}
$$

Let us give a simple example of the use of the pointfree transform, starting from the pointwise definition of injectivity of a relation R.

$$\langle \forall\, b, a, a'\ ::\ b\,R\,a\ \wedge\ b\,R\,a' \Rightarrow a = a' \rangle$$

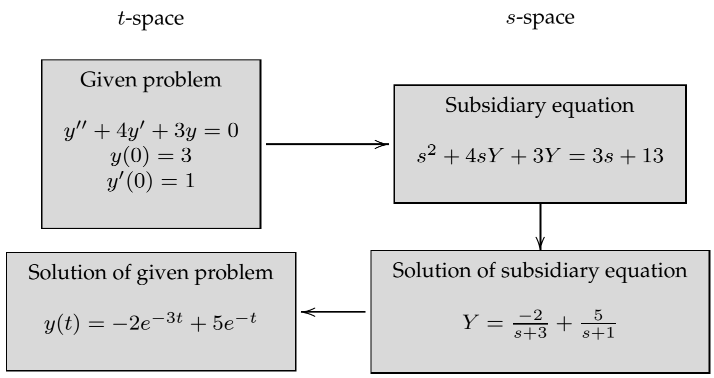

Figure 2.2: Example of Laplace transformation

By rules of quantification [12], we transform the previous expression to

$$\langle \forall\, b \ :: \ \langle \forall\, a, a' \ : \ b\, R\, a \ \wedge \ b\, R\, a' : \ a = a' \rangle \rangle$$

Again by rules of quantification [12] and additionally by converse, we arrive to the following expression:

$$\langle \forall\, a, a' \ : \ \langle \exists\, b \ :: \ a\, R^{\circ}\, b \ \wedge \ b\, R\, a' \rangle : \ a = a' \rangle$$

We now apply the pointfree transform to the existentially quantified expression and obtain,

$$\langle \forall\, a, a' \ : \ a(R^{\circ} \cdot R)a' : \ a = a' \rangle$$

Again by rules of quantification [12], we transform the previous expression to

$$\langle \forall\, a, a' \ :: \ a(R^{\circ} \cdot R)a' \Rightarrow a = a' \rangle$$

and then to

$$\mathsf{ker}\, R \subseteq id$$

i.e. the pointfree definition of injectivity of a relation R.

Two more elaborate examples of application of pointfree transform will be given in the sequel: the calculation of the pointfree definition of the relation of algorithmic refinement and that of strong bisimulation, in chapters 5 and 6, respectively.

2.4 Summary

In this chapter we have presented the essentials of binary relational algebra, which is also referred to as the *algebra of programming* [29, 12].

We may use it in the pointwise or in the pointfree style. The pointfree style makes the calculations easier. The pointfree transform is a collection of rules which help in converting a first order logic formula or pointwise relational expression to the pointfree format.

Chapter 3

Transposing Relations

3.1 Introduction

This chapter is concerned with techniques for functional transposition of binary relations. By functional transposition we mean the *faithful* representation of a relation by a (total) function. But — what is the purpose of such a representation?

Functions are well-known in mathematics and computer science because of their rich theory. For instance, they can be dualized (as happens e.g. with the projection/ injection functions), they can be Galois connected (as happens e.g. with inverse functions) and they can be parametrically polymorphic. In the latter case, they exhibit theorems "for free" [140] which can be inferred solely by inspection of their types.

However, (total) functions are not enough. In many situations, functions are *partial* in the sense that they are undefined for some of their input data. Programmers have learned to deal with this situation by enriching the codomain of such functions with a special error mark indicating that *nothing* is output. In C/C++, for instance, this leads to functions which output *pointers* to values rather than just values. In functional languages such as Haskell [78], this leads to functions which output $Maybe$-values rather than values, where $Maybe$ is datatype $Maybe\ a = Nothing \mid Just\ a$.

Partial functions are still not enough because one very often wants to describe *what* is required of a function rather than prescribe *how* the function should compute its result. A well-known example is *sorting*: sorting a list amounts to finding an ordered permutation of the list *independently* of the particular sorting algorithm eventually chosen to perform the task (eg. quicksort, mergesort, etc.). So one is concerned not only with *implementations* but also with *specifications*, which can be vague (eg. which square root is meant when one writes "$\sqrt{x}$"?) and non-deterministic. Functional programmers have learned to cope with (bounded) non-determinism by structuring the codomain of such functions as *sets* or *lists* of values.

In general, such powerset valued functions are models of binary relations: for each such f one may define the binary relation R such that bRa means exactly $b \in (f\ a)$ for all suitably typed a and b. Such R is unique for the given f. Conversely, any binary relation R is *uniquely* transposed into a set-valued function f. The existence and uniqueness of such a transformation leads to the identification of a *transpose* operator

Λ [29] satisfying the following *universal property*,

$$f = \Lambda\, R \quad \equiv \quad (bRa \quad \equiv \quad b \in f\ a) \tag{3.1}$$

for all R from A to B and $f : A \longrightarrow \mathcal{P}(B)$.

The power-transpose operator Λ establishes a well-known isomorphism between relations and set-valued functions which is often exploited in the algebra of relations, see for instance textbook [29]. Less popular and usually not identified as a transpose is the conversion of a partial function into a $Maybe$-valued function, for which one can identify, by analogy with (3.1), isomorphism Γ defined by (for all suitably typed a and b)

$$f = \Gamma\, R \quad \equiv \quad (bRa \quad \equiv \quad (f\ a = Just\ b)) \tag{3.2}$$

where R ranges over partial functions.

Terms *total* and *partial* are avoided in relation algebra because they clash with a different meaning in the context of *partial orders* and *total orders*, which are other special cases of relations. Instead, one writes *entire* for *total*, and *simple relation* is written instead of *partial function* — recall Figure 2.1, where the word *function* is reserved for total, simple relations which find a central place in the taxonomy of binary relations.

3.2 Related work

In the literature, equations (3.1) and (3.2) have been dealt with in disparate contexts. While (3.1) is adopted as “the” standard transpose in [29], for instance, (3.2) is studied in [57] as an example of an *adjunction* between the categories of total and partial functions. From the literature on the related topic of *generic membership* we select [29] and [70].

3.3 A study of generic transposition

Thanks to rule (2.6), it is easy to remove variables b and a from transposition rules (3.1) and (3.2), yielding

$$f = \Lambda\, R \quad \equiv \quad (R \quad = \quad \in \cdot f) \tag{3.3}$$

$$f = \Gamma\, R \quad \equiv \quad (R \quad = \quad i_1^\circ \cdot f) \tag{3.4}$$

where, in the second equivalence, R ranges over simple relations and $Just$ is replaced by injection i_1 associated with relator $\mathsf{Id} + 1$. In turn, f and R can also be abstracted from (3.3,3.4) using the same rule, whereby we end up with $\Lambda \ = \ (\in\cdot)^\circ$ and $\Gamma \ = \ (i_1^\circ\cdot)^\circ$.

The generalization of both equations starts from the observation that, in the same way $\in$ is the membership relation associated with the powerset, i_1° is the membership relation associated with $\mathsf{Id} + 1$, as can be easily checked:

$$
\begin{array}{cll}
& \in_{\mathsf{Id}+1} & \\
= & \quad \{ \text{ by (2.30) } \} & \\
& [\in_{\mathsf{Id}}, \in_1] & \\
= & \quad \{ \text{ by (2.28) and (2.27) } \} & \\
& [id, \perp] & (3.5) \\
= & \quad \{ \text{ by (2.25) and properties of } \perp \} & \\
& id \cdot i_1^\circ & \\
= & \quad \{ \text{ identity } \} & \\
& i_1^\circ & (3.6)
\end{array}
$$

This suggests the definitions and results which follow.

Definition 3.1 (Transposition) *Given a relator* F *with membership relation* $\in_{\mathsf{F}}$*, a particular class of binary relations* $A \xleftarrow{R} B$ *is said to be* F-transposable *iff, for each such* R*, there exists a unique function* $f : B \longrightarrow \mathsf{F}A$ *such that* $\in_{\mathsf{F}} \cdot f = R$ *holds. This is equivalent (by skolemisation) to saying that there exists a function* Γ_{F} *(called the* F-transpose*) such that, for all such* R *and* f*,*

$$
f = \Gamma_{\mathsf{F}} R \quad \equiv \quad \in_{\mathsf{F}} \cdot f = R \qquad \textit{cf. diagram} \qquad
\begin{tikzcd}
A & B \arrow[l, "R"'] \arrow[dl, "f"] \\
\mathsf{F}\,A \arrow[u, "\in_{\mathsf{F}}"] &
\end{tikzcd}
\tag{3.7}
$$

In other words, such a generic F-transpose operator is the converse of membership post-composition, as can be inferred from (3.7) by removing variables f and R:

$$\Gamma_{\mathsf{F}} \quad = \quad (\in_{\mathsf{F}} \cdot)^\circ \tag{3.8}$$

The two instances we have seen of (3.7) are the power-transpose ($\mathsf{F}\,A = \mathcal{P}(A)$) and the $Maybe$-transpose ($\mathsf{F}\,A = A + 1$). While the former is known to be applicable to every relation [29], the latter is only applicable to simple relations, a result to be justified after we review the main properties of generic transposition. These extend those presented in [29] for the power-transpose.

Properties. Cancellation and reflection

$$\in_{\mathsf{F}} \cdot \Gamma_{\mathsf{F}} R \quad = \quad R \tag{3.9}$$

$$\Gamma_{\mathsf{F}} \in_{\mathsf{F}} \quad = \quad id \tag{3.10}$$

arise from (3.7) by substitutions $f := \Gamma_{\mathsf{F}} R$ and $f := id$, respectively. Fusion

$$\Gamma_{\mathsf{F}}(T \cdot S) \quad = \quad (\Gamma_{\mathsf{F}} T) \cdot S \quad \Leftarrow \quad (\Gamma_{\mathsf{F}} T) \cdot S \text{ is a function} \tag{3.11}$$

arises from (3.7) in the same way — this time for substitution $f := (\Gamma_{\mathsf{F}} T) \cdot S$ — as follows (assuming the side condition ensuring that $(\Gamma_{\mathsf{F}} T) \cdot S$ is a function):

$$\begin{aligned}
(\Gamma_{\mathsf{F}} T) \cdot S = \Gamma_{\mathsf{F}} R \quad &\equiv \quad \in_{\mathsf{F}} \cdot ((\Gamma_{\mathsf{F}} T) \cdot S) = R \\
&\equiv \qquad \{ \text{ associativity } \} \\
&\quad\;\; (\in_{\mathsf{F}} \cdot \Gamma_{\mathsf{F}} T) \cdot S = R \\
&\equiv \qquad \{ \text{ cancellation (3.9) } \} \\
&\quad\;\; T \cdot S = R
\end{aligned}$$

The side condition of (3.11) requires S to be entire but not necessarily simple. In fact, it suffices that $\mathit{img}\, S \subseteq \mathit{ker}\, (\Gamma_{\mathsf{F}} T)$ since, in general, the simplicity of $f \cdot S$ equivales $\mathit{img}\, S \subseteq \mathit{ker}\, f$:

$$\begin{aligned}
& \mathit{img}\, S \subseteq \mathit{ker}\, f \\
\equiv & \qquad \{ \text{ definitions } \} \\
& (S \cdot S^\circ) \subseteq f^\circ \cdot f \\
\equiv & \qquad \{ \; id \text{ is the unit of composition } \} \\
& (S \cdot S^\circ) \subseteq f^\circ \cdot id \cdot f \\
\equiv & \qquad \{ \text{ shunting rules (2.15) } \} \\
& f \cdot (S \cdot S^\circ) \cdot f^\circ \subseteq id \\
\equiv & \qquad \{ \text{ composition is associative ; converse of composition } \} \\
& (f \cdot S) \cdot (f \cdot S)^\circ \subseteq id \\
\equiv & \qquad \{ \text{ definition of } \mathit{img} \} \\
& \mathit{img}\, (f \cdot S) \subseteq id \\
\equiv & \qquad \{ \text{ simplicity } \} \\
& (f \cdot S) \text{ is simple}
\end{aligned}$$

In summary, the simplicity of (entire) S is a sufficient (but not necessary) condition for the fusion law (3.11) to hold. In this case, S is a function, and it is under this condition that the law is presented in [29] [1].

Substitution $f := \Gamma_{\mathsf{F}} S$ in (3.7) and cancellation (3.9) lead to the *injectivity law*,

$$\Gamma_{\mathsf{F}} S = \Gamma_{\mathsf{F}} R \quad \equiv \quad S = R \tag{3.12}$$

Finally, the generic version of the *absorption property*,

$$\mathsf{F}\, R \cdot \Gamma_{\mathsf{F}} S = \Gamma_{\mathsf{F}}(R \cdot S) \quad \Leftarrow \quad R \cdot \in_{\mathsf{F}} \subseteq \in_{\mathsf{F}} \cdot \mathsf{F}\, R \tag{3.13}$$

is justified as follows:

$$
\begin{array}{ll}
& \mathsf{F}\,R \cdot \Gamma_{\mathsf{F}} S = \Gamma_{\mathsf{F}}(R \cdot S) \\
\equiv & \qquad \{\ \text{universal property (3.7)}\ \} \\
& \in_{\mathsf{F}} \cdot \mathsf{F}\,R \cdot \Gamma_{\mathsf{F}} S = R \cdot S \\
\equiv & \qquad \{\ \text{assume } \in_{\mathsf{F}} \cdot \mathsf{F}\,R = R \cdot \in_{\mathsf{F}})\ \} \\
& R \cdot \in_{\mathsf{F}} \cdot \Gamma_{\mathsf{F}} S = R \cdot S \\
\equiv & \qquad \{\ \text{cancellation (3.9)}\ \} \\
& R \cdot S = R \cdot S
\end{array}
$$

The side condition of (3.13) arises from the property assumed in the second step of the proof. Together with (2.26), it establishes the required equality by anti-symmetry.

Unit and inclusion. Two concepts of set-theory can be made generic in the context above. The first one has to do with *singletons*, that is, data structures which contain a single datum. The function τ_{F} mapping every A to its singleton of type F is obtainable by transposing id, $\tau_{\mathsf{F}} = \Gamma_{\mathsf{F}} id$, and is such that

$$\in_{\mathsf{F}} \cdot \tau_{\mathsf{F}} = id$$

via (3.7) and such that

$$\tau_{\mathsf{F}} \cdot f = \Gamma_{\mathsf{F}} f$$

by the fusion law.

Two obvious instances of unit are

$$\tau_P\ x = \{x\}$$

concerning the powerset and

$$\tau_{Maybe}\ x = Just\ x$$

concerning $Maybe$.

Another concept relevant in the sequel is *generic inclusion*, defined by

$$\mathsf{F}\,A \xleftarrow{\in_{\mathsf{F}} \backslash \in_{\mathsf{F}}} \mathsf{F}\,A \tag{3.14}$$

and involving *left division* (2.15), the relational operator which is defined by the fact that $(R \setminus\ )$ is the upper-adjoint of $(R\, \cdot\ )$ for every R. For the powerset relator, (3.14) defines set inclusion:

$$x(\in_{\mathsf{F}} \setminus \in_{\mathsf{F}})y \equiv \langle \forall\ c\ :\ c \in x :\ c \in y \rangle$$

3.4 Instances of generic transposition

In this section we discuss the power-transpose ($\mathsf{F} = \mathcal{P}()$) and the $Maybe$-transpose ($\mathsf{F} = \mathsf{Id} + 1$) as instances of the generic transpose (3.7). Unlike the former, the latter is not applicable to every relation. To conclude that only simple relations are $Maybe$-transposable, we first show that, for every F-transposable R, its image is at most the image of $\in_{\mathsf{F}}$:

$$img\, R \quad \subseteq \quad img\ \in_{\mathsf{F}} \tag{3.15}$$

The proof is easy to follow:

$$\begin{array}{cl} & img\, R \\ = & \quad \{\ \text{definition}\ \} \\ & R \cdot R^{\circ} \\ = & \quad \{\ R \text{ is } \mathsf{F}\text{-transposable}\ ;\ \text{cancellation (3.9)}\ \} \\ & (\in_{\mathsf{F}} \cdot \Gamma_{\mathsf{F}} R) \cdot (\in_{\mathsf{F}} \cdot \Gamma_{\mathsf{F}} R)^{\circ} \\ = & \quad \{\ \text{converses}\ \} \\ & \in_{\mathsf{F}} \cdot \Gamma_{\mathsf{F}} R \cdot (\Gamma_{\mathsf{F}} R)^{\circ} \cdot {\in_{\mathsf{F}}}^{\circ} \\ \subseteq & \quad \{\ \Gamma_{\mathsf{F}} R \text{ is simple}\ ;\ \text{monotonicity}\ \} \\ & \in_{\mathsf{F}} \cdot {\in_{\mathsf{F}}}^{\circ} \\ = & \quad \{\ \text{definition}\ \} \\ & img\ \in_{\mathsf{F}} \end{array}$$

So, $\in_{\mathsf{F}}$ restricts the class of relations R which are F-*transposable*. Concerning the power-transpose, it is easy to see that $img\ \in_{\mathsf{F}} = \top$ since, for every a, a', there exists at least the set $\{a, a'\}$ which both a and a' belong to. Therefore, no restriction is imposed on $img\, R$ and transposition witnesses the well-known isomorphism $(2^A)^B \cong 2^{B \times A}$ (writing 2^A for $\mathcal{P}(A)$ and identifying every relation with its *graph*, a set of pairs).

By contrast, simple memberships can only be associated to the transposition of simple relations. This is what happens with $\in_{\mathsf{Id}+1} = i_1^{\circ}$ which, as the converse of an injection, is simple (2.5).

Conversely, appendix B.1 shows that all simple relations are $(\mathsf{Id} + 1)$-transposable. Therefore, $(\mathsf{Id} + 1)$-transposability *defines* the class of simple relations and witnesses isomorphism

$$(B + 1)^A \quad \cong \quad A \rightharpoonup B \tag{3.16}$$

where $A \rightharpoonup B$ denotes the set of all simple relations from A to B. This isomorphism is central to the data refinement calculus studied in chapter 4.

Another difference between the two instances of generic transposition considered so far can be found in the application of the absorption property (3.13). That its side condition holds for the $Maybe$-transpose is easy to show:

$$
\begin{array}{cl}
 & R \cdot i_1^\circ \subseteq i_1^\circ \cdot (R + id) \\
\equiv & \quad \{ \text{ shunting } \} \\
 & i_1 \cdot R \subseteq (R + id) \cdot i_1 \\
\equiv & \quad \{ \; R + S \text{ (2.23) is a coproduct [29] } \} \\
 & i_1 \cdot R \subseteq i_1 \cdot R \\
\equiv & \quad \{ \text{reflexivity} \} \\
 & True
\end{array}
$$

Concerning the power-transpose, [29] defines the absorption property for the *existential image* functor, $\mathsf{E}R \;=\; \Lambda(R\cdot \in)$, which coincides with the powerset relator for functions. So, the absorption property of the power-transpose can only be used where R is a function: $\mathcal{P}(f) \cdot \Lambda S = \Lambda(f \cdot S)$.

Finally, inclusion (3.14) for the power-transpose is the set-theoretic *subset ordering* recall (3.14), while its $Maybe$ instance corresponds to the expected *flat-cpo* ordering":

$$x(\in_{\mathsf{Id}+1} \setminus \in_{\mathsf{Id}+1})y \quad \equiv \quad \forall a.\; x = (i_1\; a) \quad \Rightarrow \quad y = (i_1\; a)$$

So $Nothing$ will be included in anything and every "non-$Nothing$" x will be included only in itself [2].

3.5 Applications of generic transpose

The main purpose of representing relations by functions is to take advantage of the (sub)calculus of functions when applied to the transposed relations. In particular, transposition can be used to infer properties of relational combinators. Suppose that $f \oplus g$ is a functional combinator whose properties are known, for instance, $f \oplus g = [f, g]$ for which we know universal property

$$k = [f, g] \quad \equiv \quad \begin{cases} k \cdot i_1 = f \\ k \cdot i_2 = g \end{cases} \tag{3.17}$$

We may inquire about the corresponding property of another, this time *relational*, combinator $R \otimes S$ induced by transposition:

$$\Gamma_{\mathsf{F}}(R \otimes S) \quad = \quad (\Gamma_{\mathsf{F}} R) \oplus (\Gamma_{\mathsf{F}} S) \tag{3.18}$$

$$\equiv \qquad \{ \text{ (3.7) } \}$$

$$R \otimes S \quad = \quad \in_{\mathsf{F}} \cdot ((\Gamma_{\mathsf{F}} R) \oplus (\Gamma_{\mathsf{F}} S)) \tag{3.19}$$

This can happen in essentially two ways, which are described next.

Proof of universality by transposition. It may happen that the universal property of functional combinator $\oplus$ is carried intact along the move from functions to relations. A good example of this is relational coproduct, whose existence is shown in [29] to stem from functional coproducts (3.17) by transposition [3]. One only has to instantiate (3.17) for $k, f, g := \Gamma_{\mathsf{F}}T, \Gamma_{\mathsf{F}}R, \Gamma_{\mathsf{F}}S$ and reason:

$$
\begin{array}{cl}
& \Gamma_{\mathsf{F}}T = [\Gamma_{\mathsf{F}}R, \Gamma_{\mathsf{F}}S] \ \equiv \ (\Gamma_{\mathsf{F}}T) \cdot i_1 = \Gamma_{\mathsf{F}}R \ \wedge \ (\Gamma_{\mathsf{F}}T) \cdot i_2 = \Gamma_{\mathsf{F}}S \\
\equiv & \quad \{ \text{ (3.7) and fusion (3.11) twice, for } S := i_1, i_2 \ \} \\
& T = \ \in \cdot [\Gamma_{\mathsf{F}}R, \Gamma_{\mathsf{F}}S] \ \equiv \ \Gamma_{\mathsf{F}}(T \cdot i_1) = \Gamma_{\mathsf{F}}R \ \wedge \ \Gamma_{\mathsf{F}}(T \cdot i_2) = \Gamma_{\mathsf{F}}S \\
\equiv & \quad \{ \text{ injectivity (3.12) } \} \\
& T = \ \in \cdot [\Gamma_{\mathsf{F}}R, \Gamma_{\mathsf{F}}S] \ \equiv \ T \cdot i_1 = R \ \wedge \ T \cdot i_2 = S \\
\equiv & \quad \{ \text{ introduce notation } [R, S] = \ \in \cdot [\Gamma_{\mathsf{F}}R, \Gamma_{\mathsf{F}}S] \ \} \\
& T = [R, S] \ \equiv \ T \cdot i_1 = R \ \wedge \ T \cdot i_2 = S \\
\equiv & \quad \{ \text{ coproduct definition } \} \\
& [R, S] \text{ defines a coproduct}
\end{array}
$$

Defined in this way, relational coproducts enjoy all properties of functional coproducts, namely fusion, absorption etc.

This calculation, however, cannot be dualized to the generalization of the *split*-combinator $\langle f, g \rangle$ to relational $\langle R, S \rangle$. In fact, relational product is not a categorical product, which means that some properties will not hold, namely the fusion law,

$$\langle g, h \rangle \cdot f = \langle g \cdot f, h \cdot f \rangle \tag{3.20}$$

when g, h, f are replaced by relations. According to [29], what we have is

$$\langle R, S \rangle \cdot f = \langle R \cdot f, S \cdot f \rangle \tag{3.21}$$

whose proof can be carried out by resorting to the explicit definition of the *split* combinator (2.24) and some properties of simple relations grounded on the so-called *modular law* [4].

In the following we present an alternative proof of (3.21) as an example of the calculation power of transposes *combined* with *Reynolds abstraction theorem* in the pointfree style [10]. The proof is more general and leads to other versions of the law, depending upon which transposition is adopted, that is, which class of relations is considered.

From the type of functional *split*,

$$\langle \_, \_ \rangle \quad : \quad ((A \times B) \leftarrow C) \leftarrow ((A \leftarrow C) \times (B \leftarrow C)) \tag{3.22}$$

we want to define the relational version of this combinator — let us denote it by $(\_ \otimes \_)$ for the time being — via the adaptation of $\langle \_, \_ \rangle$ (3.22) to transposed relations, to be denoted by $(\_ \oplus \_)$. This will be of type

$$t \quad = \quad (\mathsf{F}\,(A \times B) \leftarrow C) \leftarrow ((\mathsf{F}\,A \leftarrow C) \times (\mathsf{F}\,B \leftarrow C)) \tag{3.23}$$

Reynolds abstraction theorem. Instead of defining $(\_ \oplus \_)$ explicitly, we will reason about its properties by applying the *abstraction theorem* due to J. Reynolds [125] and advertised by P. Wadler [140] under the "*theorem for free*" heading. We follow the pointfree styled presentation of this theorem in [10], which is remarkably elegant: let f be a polymorphic *function* f of type t, that is, $f \;:\; t$, whose type t can be written according to the following "grammar" of types:

$$\begin{array}{lll}
t & ::= & t' \leftarrow t'' \\
t & ::= & \mathsf{F}(t_1, \ldots, t_n) \qquad \text{for } n\text{-ary relator } \mathsf{F} \\
t & ::= & v \qquad \text{for } v \text{ a type variable (= polymorphism "dimension")}
\end{array}$$

Let V be the set of type variables involved in type t; $\{R_v\}_{v \in V}$ be a V-indexed family of relations (f_v in case all such R_v are functions); and R_t be a relation defined inductively as follows:

$$\begin{array}{rcl}
R_{t:=\mathsf{F}(t_1,\ldots,t_n)} & = & \mathsf{F}(R_{t_1}, \ldots, R_{t_n}) \\
R_{t:=v} & = & R_v \\
R_{t:=t' \leftarrow t''} & = & R_{t'} \leftarrow R_{t''}
\end{array}$$

where $R_{t'} \leftarrow R_{t''}$ is defined by (2.9). The *free theorem of type t* reads as follows: *given any function $f : t$ and V as above, $f\ R_t\ f$ holds for any relational instantiation of type variables in V.* Note that this theorem is a result about t and holds for *any* polymorphic function of type t *independently* of its actual definition [5].

In the remainder of this section we deduce the *free theorem* of type in t (3.23) and draw conclusions about the fusion and absorption properties of relational split based on such a theorem. First we calculate R_t:

$$\begin{array}{cl}
 & R_t \\
\equiv & \qquad \{ \text{ induction on the structure of } t \text{ (3.23) } \} \\
 & (\mathsf{F}\,(R_A \times R_B) \leftarrow R_C) \leftarrow ((\mathsf{F}\,R_A \leftarrow R_C) \times (\mathsf{F}\,R_B \leftarrow R_C)) \\
\equiv & \qquad \{ \text{ substitution } R_A, R_B, R_C := R, S, Q \text{ in order to remove subscripts } \} \\
 & (\mathsf{F}\,(R \times S) \leftarrow Q) \leftarrow ((\mathsf{F}\,R \leftarrow Q) \times (\mathsf{F}\,S \leftarrow Q))
\end{array}$$

Next we calculate the free theorem of $(\_ \oplus \_) : t$:

$$
\begin{array}{ll}
& (\_ \oplus \_)(R_t)(\_ \oplus \_) \\
= & \quad \{ \text{ expansion of } R_t \} \\
& (\_ \oplus \_)(\mathsf{F}\,(R \times S) \leftarrow Q) \leftarrow ((\mathsf{F}\,R \leftarrow Q) \times (\mathsf{F}\,S \leftarrow Q)))(\_ \oplus \_) \\
= & \quad \{ \text{ meaning of Reynolds arrow combinator (2.9) } \} \\
& (\_ \oplus \_) \cdot ((\mathsf{F}\,R \leftarrow Q) \times (\mathsf{F}\,S \leftarrow Q)) \quad \subseteq \quad \mathsf{F}\,(R \times S) \leftarrow Q) \cdot (\_ \oplus \_) \\
= & \quad \{ \text{ } \textit{shunting} \text{ (2.15) } \} \\
& (\mathsf{F}\,R \leftarrow Q) \times (\mathsf{F}\,S \leftarrow Q) \quad \subseteq \quad (\_ \oplus \_)^{\circ} \cdot (\mathsf{F}\,(R \times S) \leftarrow Q) \cdot (\_ \oplus \_) \\
= & \quad \{ \text{ going pointwise and (2.6) } \} \\
& (f, g)((\mathsf{F}\,R \leftarrow Q) \times (\mathsf{F}\,S \leftarrow Q))(h, k) \quad \Rightarrow \quad (f \oplus g)(\mathsf{F}\,(R \times S) \leftarrow Q)(h \oplus k) \\
= & \quad \{ \text{ product relator and (2.9) } \} \\
& f(\mathsf{F}\,R \leftarrow Q)h \ \wedge \ g(\mathsf{F}\,S \leftarrow Q)k \quad \Rightarrow \quad (f \oplus g) \cdot Q \quad \subseteq \quad \mathsf{F}\,(R \times S) \cdot (h \oplus k) \\
= & \quad \{ \text{ Reynolds arrow combinator (2.9) three times } \} \\
& f \cdot Q \subseteq \mathsf{F}\,R \cdot h \quad \wedge \quad g \cdot Q \subseteq \mathsf{F}\,S \cdot k \quad \Rightarrow \quad (f \oplus g) \cdot Q \quad \subseteq \quad \mathsf{F}\,(R \times S) \cdot (h \oplus k)
\end{array}
$$

Should we replace functions f, h, g, k by transposed relations $\Gamma_{\mathsf{F}}U, \Gamma_{\mathsf{F}}V, \Gamma_{\mathsf{F}}X, \Gamma_{\mathsf{F}}Z$, respectively, we obtain

$$((\Gamma_{\mathsf{F}}U) \oplus (\Gamma_{\mathsf{F}}X)) \cdot Q \quad \subseteq \quad \mathsf{F}\,(R \times S) \cdot ((\Gamma_{\mathsf{F}}V) \oplus (\Gamma_{\mathsf{F}}Z)) \tag{3.24}$$

provided conjunction

$$(\Gamma_{\mathsf{F}}U) \cdot Q \subseteq \mathsf{F}\,R \cdot (\Gamma_{\mathsf{F}}V) \quad \wedge \quad (\Gamma_{\mathsf{F}}X) \cdot Q \subseteq \mathsf{F}\,S \cdot (\Gamma_{\mathsf{F}}Z) \tag{3.25}$$

holds. Assuming (3.18), (3.24) can be re-written as

$$\Gamma_{\mathsf{F}}(U \otimes X) \cdot Q \quad \subseteq \quad \mathsf{F}\,(R \times S) \cdot \Gamma_{\mathsf{F}}(V \otimes Z) \tag{3.26}$$

At this point we restrict Q to a function q and apply the fusion law (3.11) without extra side conditions:

$$\Gamma_{\mathsf{F}}((U \otimes X) \cdot q) \quad \subseteq \quad \mathsf{F}\,(R \times S) \cdot \Gamma_{\mathsf{F}}(V \otimes Z) \tag{3.27}$$

For $R, S := id, id$ we will obtain —"for free" — the standard fusion law

$$(U \otimes X) \cdot q \quad = \quad (U \cdot q \otimes X \cdot q)$$

presented in [29] for the *split* combinator (3.21), ie. for $(R \otimes S) = \langle R, S \rangle$:

$$\langle R, S \rangle \cdot q \quad = \quad \langle R \cdot q, S \cdot q \rangle \tag{3.28}$$

In the reasoning, all factors involving R and S disappear and fusion takes place in both conjuncts of (3.25). Moreover, inclusion ($\subseteq$) becomes equality of transposed relations — thanks to (2.21) — and injectivity (3.12) is used to remove all occurrences of Γ_{F}. Wherever R and S are not identities, one has different results depending on the behaviour of the chosen transposition concerning the absorption property (3.13).

$Maybe$ **transpose.** In case of *simple* relations under the $Maybe$-transpose, absorption has no side condition, and so (3.27) rewrites to

$$(U \otimes X) \cdot q \quad = \quad (R \times S) \cdot (V \otimes Z) \tag{3.29}$$

by further use of (2.21) — recall that transposed relations are functions — and injectivity (3.12), provided (3.25) holds, which boils down to $U \cdot q = R \cdot V$ and $X \cdot q = S \cdot Z$ under a similar reasoning. For $q := id$ and $(\_ \otimes \_)$ instantiated to relational split, this becomes absorption law

$$\langle R \cdot V, S \cdot Z \rangle \quad = \quad (R \times S) \cdot \langle V, Z \rangle \qquad \text{if} \quad R, S, V, Z \text{ are simple} \tag{3.30}$$

In summary, our reasoning has shown that the *absorption* law for *simple* relations is a free theorem.

Power transpose. In case of arbitrary relations under the power-transpose, absorption requires R and S in (3.27) to be functions (say r, s), whereby the equation re-writes to

$$\Gamma_{\mathsf{F}}((U \otimes X) \cdot q) \quad \subseteq \quad \Gamma_{\mathsf{F}}((r \times s) \cdot (V \otimes Z)) \tag{3.31}$$

provided $\Gamma_{\mathsf{F}}(U \cdot q) \subseteq \Gamma_{\mathsf{F}}(r \cdot V)$ and $\Gamma_{\mathsf{F}}(X \cdot q) \subseteq \Gamma_{\mathsf{F}}(s \cdot Z)$ hold. Again by combined use of (2.21) and injectivity (3.12) one gets

$$(U \otimes X) \cdot q \quad = \quad (r \times s) \cdot (V \otimes Z) \tag{3.32}$$

provided $U \cdot q = r \cdot V$ and $X \cdot q = s \cdot Z$ hold. Again instantiating $q := id$ and $(\_ \otimes \_) = \langle \_, \_ \rangle$, this becomes absorption law

$$\langle r \cdot V, s \cdot Z \rangle \quad = \quad (r \times s) \cdot \langle V, Z \rangle \tag{3.33}$$

Bird and Moor [29] show, in (admittedly) a rather tricky way, that product absorption holds for *arbitrary* relations. Our calculations have identified two restricted versions of such a law — (3.30) and (3.33) — as "free" theorems, which could be deduced in a more elegant, *parametric* way.

3.6 Generic monads from transposes and memberships

The fact that both the *powerset* and the *Maybe* relators involved in (3.3) and (3.4) are both *monads* is sympthomatic of a close relationship which exists between monads and transposition. We conclude this chapter by discussing such a connection in detail. Let us first introduce the definition of a monad, as in [82].

Definition 3.1 *A monad* $\mathsf{F} = (\mathsf{F}, u, \mu)$ *in a category* X *consists of an endofunctor* $\mathsf{F} : X \to X$ *and two natural transformations,* unit

$$\begin{tikzcd} A \arrow[d, "f"] \\ B \end{tikzcd} \qquad \begin{tikzcd} F\ A \arrow[d, "F\ f"'] & A \arrow[l, "u"'] \arrow[d, "f"] \\ F\ B & B \arrow[l, "u"] \end{tikzcd} \tag{3.34}$$

and multiplication

$$\begin{tikzcd} A \arrow[d, "f"] \\ B \end{tikzcd} \qquad \begin{tikzcd} F\ A \arrow[d, "F\ f"'] & F^2\ A \arrow[l, "\mu"'] \arrow[d, "F^2\ f"] \\ F\ B & F^2\ B \arrow[l, "\mu"] \end{tikzcd} \tag{3.35}$$

which satisfy the following equations:

$$\mu \cdot u = \mu \cdot \mathsf{F}\, u = id \tag{3.36}$$

$$\mu \cdot \mu = \mu \cdot F\mu \tag{3.37}$$

cf. diagrams

$$\begin{tikzcd} F^2\ A \arrow[d, "\mu"'] & F\ A \arrow[l, "u"'] \arrow[d, "F\ u"] \arrow[dl, "id"] \\ F\ A & F^2\ A \arrow[l, "\mu"] \end{tikzcd}$$

and

$$\begin{tikzcd} F^2\ A \arrow[d, "\mu"'] & F^3\ A \arrow[l, "\mu"'] \arrow[d, "F\ \mu"] \\ F\ A & F^2\ A \arrow[l, "\mu"] \end{tikzcd}$$

respectively. Let Γ_{F} be the generic transpose as defined by (3.7). Let us define

$$\mu \;=\; \Gamma_{\mathsf{F}}(\in_{\mathsf{F}} \cdot \in_{\mathsf{F}}) = \Gamma_{\mathsf{F}} \in_{\mathsf{F}^2} \tag{3.38}$$

$$u \;=\; \tau_{\mathsf{F}} = \Gamma_{\mathsf{F}} id \tag{3.39}$$

which are candidates to form a monad. Membership $\in_{\mathsf{F}}$ is a natural transformation,

$$f \cdot \in_{\mathsf{F}} \;=\; \in_{\mathsf{F}} \cdot \mathsf{F}\, f \tag{3.40}$$

an instance of (2.26).

By cancellation we obtain

$$\begin{aligned} \in_{\mathsf{F}} \cdot u &= id & (3.41) \\ \in_{\mathsf{F}} \cdot \mu &= \in_{\mathsf{F}} \cdot \in_{\mathsf{F}} & (3.42) \end{aligned}$$

We also have

$$\mu \cdot (\mathsf{F}\, f) = \Gamma_F(\in_{\mathsf{F}} \cdot f \cdot \in_{\mathsf{F}}) \qquad (3.43)$$

cf.

$$\begin{array}{ll} & \mu \cdot \mathsf{F}\, f \\ = & \qquad \{ \text{ definition (3.38)} \} \\ & \Gamma_F(\in_{\mathsf{F}} \cdot \in_{\mathsf{F}}) \cdot \mathsf{F}\, f \\ = & \qquad \{ \text{ fusion (3.11) since } \mathsf{F}\, f \text{ is a function} \} \\ & \Gamma_F(\in_{\mathsf{F}} \cdot \in_{\mathsf{F}} \cdot \mathsf{F}\, f) \\ = & \qquad \{ \text{ by (3.40)} \} \\ & \Gamma_F(\in_{\mathsf{F}} \cdot f \cdot \in_{\mathsf{F}}) \end{array}$$

The facts above are enough to show that μ and u defined by (3.38) and (3.39) form a monad:

Unit:

$$\mu \cdot u = \mu \cdot \mathsf{F}\, u = id \qquad (3.44)$$

Calculation:

$$\begin{array}{ll} & \mu \cdot u \\ = & \qquad \{ \text{ definitions (3.38) and (3.39)} \} \\ & \Gamma_F(\in_{\mathsf{F}} \cdot \in_{\mathsf{F}}) \cdot (\Gamma_F\ id) \\ = & \qquad \{ \text{ fusion (3.11) followed by cancellation (3.9) } \} \\ & \Gamma_F(\in_{\mathsf{F}} \cdot id) \\ = & \qquad \{ \text{ natural-}id \} \\ & \Gamma_F(\in_{\mathsf{F}}) \\ = & \qquad \{ \text{ generic transpose reflection (3.10) } \} \\ & id \\ = & \qquad \{ \text{ natural-}id \text{ and reflection again} \} \\ & \Gamma_F(id \cdot \in_{\mathsf{F}}) \\ = & \qquad \{ \text{ by (3.42)} \} \\ & \Gamma_F(\in_{\mathsf{F}} \cdot u \cdot \in_{\mathsf{F}}) \\ = & \qquad \{ \text{ (3.43)} \} \\ & \mu \cdot \mathsf{F}\, u \end{array}$$

Multiplication:

$$\mu \cdot \mu = \mu \cdot F\mu \tag{3.45}$$

Calculation:

$$\begin{array}{ll}
 & \mu \cdot \mu \\
= & \quad \{ \text{ definition (3.38) } \} \\
 & \Gamma_F(\in_\mathsf{F} \cdot \in_\mathsf{F}) \cdot \mu \\
= & \quad \{ \text{ fusion (3.11) } \} \\
 & \Gamma_F((\in_\mathsf{F} \cdot \in_\mathsf{F}) \cdot \mu) \\
= & \quad \{ \text{ associativity} \} \\
 & \Gamma_F(\in_\mathsf{F} \cdot (\in_\mathsf{F} \cdot \mu)) \\
= & \quad \{ \text{ (3.42}\} \\
 & \Gamma_F(\in_\mathsf{F} \cdot (\in_\mathsf{F} \cdot \in_\mathsf{F})) \\
= & \quad \{ \text{ associativity} \} \\
 & \Gamma_F((\in_\mathsf{F} \cdot \in_\mathsf{F}) \cdot \in_\mathsf{F}) \\
= & \quad \{ \text{ (3.42}\} \\
 & \Gamma_F((\in_\mathsf{F} \cdot \mu) \cdot \in_\mathsf{F}) \\
= & \quad \{ \text{ (3.43}\} \\
 & \mu \cdot F\mu
\end{array}$$

In summary, we conclude that the relator F involved in generic transposition *always* is a monad, whose unit and multiplication are solely defined in terms of membership $\in_\mathsf{F}$ and Γ_F.

3.7 Summary

Functional transposition is a technique for converting relations into functions aimed at developing the algebra of binary relations indirectly *via* the algebra of functions. A functional transpose of a binary relation of a particular class is an “F-resultric” function where F is a parametric datatype with membership. This chapter develops a basis for a theory of *generic transposition* under the following slogan: *generic transpose is the converse of membership post-composition*.

Instances of *generic transpose* provide universal properties which all inhabitants of particular *classes* of relations satisfy. Two such instances are considered in this chapter, one applicable to any relation and the other applicable only to simple relations. In either cases, *genericity* consists of reasoning about the transposed relations without using the explicit definition of the transpose operator itself.

Our illustration of the purpose of transposition takes advantage of the *free theorem* of a polymorphic function. We show how to derive laws of relational combinators as free theorems involving their transposes.

The relator F involved in generic transposition *always* is a monad, whose unit and multiplication are solely defined in terms of membership $\in_{\mathsf{F}}$ and transposition Γ_{F}.

Part II

Refinement Calculi

Chapter 4

Data Refinement

4.1 Introduction

Computer programs are made of *algorithms* which manipulate *data* — recall the title of Wirth's textbook [141] on the subject. When taking the dynamic semantics of software systems into account we realize that such systems are coalgebras [132].

The carrier of the (co)algebra represents the data dimension (i.e. data structures involved) which is manipulated by its algorithmic dimension, which combines operations in an algebra with the dynamics of a coalgebra. Specification methods such as VDM recommend that one of the above dimensions — data refinement — be dealt with first, possibly involving several iterations. Once a satisfactory *implementation data model* is reached, refinement decisions are taken on the orthogonal direction — algorithmic refinement — so as to, eventually, reach executable code. This view is in slight contrast with [95, 93], where data refinement is regarded as a special case of algoritmic refinement, being the action of replacing an abstract type by a more concrete one in a program, while preserving its algorithmic behaviour. The corresponding refinement calculus stems from an extension of Dijkstra calculus [52] of predicate transformers. Such a "calculational style" was firts introduced in [68], in a relational setting, as we saw in section 2.1. Reference [45] is among the first in the literature to characterize functional data-refinement by calculation (transformation). Reference [103] shows how to apply this strategy in the relational (*pre/post*-condition) context of the VDM methodology.

However, a slight difficulty persists in both functional and relational data refinement: one has to choose (i.e. guess) the *abstraction invariant* [93] which links abstract values to concrete values. Such an invariant, which in the functional style can be factored into a *concrete invariant* and an *abstraction function*, can be hard to formulate in practical, realistic examples. It would be preferable to be able to *calculate* such an invariant. That is to say, one needs calculi for the stepwise refinement of the data themselves. This is precisely the main target of the SETS calculus [106] which is the main subject of this chapter. Recently, [116] presents a relational evolution of [106] towards a more agile way of justifying the rules of the calculus.

We begin with an example of data refinement which is known to every computer scientist or programmer: the refinement of data collections in terms of hash tables. This example, which illustrates the need for relational reasoning in data refinement, is taken from reference [117], where it is developed in detail.

4.2 An Example of Data Refinement

Hashing. Hash tables are well known data structures [141, 72] whose purpose is to efficiently combine the advantages of both static and dynamic storage of data. Static structures such as *arrays* provide random access to data but have the disadvantage of filling too much primary storage. Dynamic, *pointer*-based structures (*eg.* search lists, search trees etc.) are more versatile with respect to storage requirements but access to data is not as immediate.

The idea of *hashing* is suggested by the informal meaning of the term itself: a large database file is "hashed" into as many "pieces" as possible, each of which is randomly accessed. Since each sub-database is smaller than the original, the time spent on accessing data is shortened by some order of magnitude. Random access is normally achieved by a so-called *hash function*, say $B \xleftarrow{h} A$, which computes, for each data item a (of type A), its *location* $h\ a$ (of type B) in the *hash table*. Standard terminology regards as *synonyms* all data competing for the same location. A set of synonyms is called a *bucket*.

Data collision can be handled either by *eg. linear probing* [141] or *overflow handling* [72]. The former is not a totally correct representation of a data collection. Overflow handling consists in partitioning a given data collection $S \subseteq A$ into n-many, disjoint buckets, each one addressed by the relevant hash index computed by h [1].

This partition can be modelled by a function t of type $\mathcal{P}(A) \xleftarrow{t} B$ and the so-called "hashing effect" is the following: the membership test $a \in S$ (which requires an inspection of the whole dataset S) can be replaced by $a \in t(h\ a)$ (which only inspects the bucket addressed by location $h\ a$). That is, equivalence

$$a \in S \quad \equiv \quad a \in t(h\ a) \tag{4.1}$$

must hold for all a for t to be regarded as a *hash table*.

Clearly, (4.1) establishes the abstraction invariant which links the abstract data structure S (a set of data) to the concrete data structure t (a hash table). Let us PF-transform this equation:

$$
\begin{array}{cl}
& a \in S \;\equiv\; a \in t(h\ a) \\
= & \quad \{\ \text{introduce } b = h\ a\ \} \\
& a \in S\ \wedge\ b = h\ a \;\equiv\; a \in (t\ b) \\
= & \quad \{\ \text{introduce } a = a'\ \} \\
& a \in S\ \wedge\ a = a'\ \wedge\ b = h\ a' \;\equiv\; a \in (t\ b) \\
= & \quad \{\ \text{regard } S \text{ as a coreflexive ; converse of hash function}\ \} \\
& aSa'\ \wedge\ a'h^{\circ}b \;\equiv\; a \in (t\ b) \\
= & \quad \{\ \text{relational composition and rule (2.6)}\ \} \\
& a(S \cdot h^{\circ})b \;\equiv\; a(\in \cdot\, t)b \\
= & \quad \{\ \text{go pointfree}\ \} \\
& S \cdot h^{\circ} \;=\; \in \cdot\, t \\
= & \quad \{\ \text{power transpose (3.1)}\} \\
& t \;=\; \Lambda(S \cdot h^{\circ})
\end{array}
$$

So, for an arbitrary coreflexive relation (representing a set) $A \xleftarrow{S} A$, its hash-transpose (for a fixed hash function $B \xleftarrow{h} A$) is a function $\mathcal{P}(A) \xleftarrow{t} B$, satisfying

$$
\in \cdot\, t \;=\; S \cdot h^{\circ} \qquad
\begin{tikzcd}
A & A \arrow[l, "S"'] \\
\mathcal{P}(A) \arrow[u, "\in"] & B \arrow[l, "t"] \arrow[u, "h^{\circ}"']
\end{tikzcd}
$$

By defining

$$\Theta_h\ S \stackrel{\text{def}}{=} \Lambda(S \cdot h^{\circ}) \tag{4.2}$$

we obtain a h-indexed family of *hash transpose* operators and associated universal properties, universally quantified in t and s:

$$t = \Theta_h\ S \quad \equiv \quad \in \cdot\, t = S \cdot h^{\circ} \tag{4.3}$$

Universal property (4.3) is interesting because it tells us how the abstaction invariant we started from leads to a definition of the *representation* function (Θ_h) which maps sets of data to *hash tables*. In fact, definition (4.2) is a consequence of (4.3), by left cancellation: just let $t = \Theta_h\ S$ in (4.3),

$$\in \cdot\ (\Theta_h\ S) \;=\; S \cdot h^{\circ} \tag{4.4}$$

and aplly (3.1):

$$\Theta_h\, S = \Lambda(S \cdot h^\circ)$$

In summary, the hash-transpose extends the power-transpose of coreflexive relations in the sense that $\Lambda = (\Theta_{id})$. Put in other words, the power-transpose is the hash-transpose using id as hash function. In practice, this is an extreme case, since some "lack of injectivity" is required of h for the hash effect to take place. Note, in passing, that the other extreme case is $h = !_A$, where $1 \xleftarrow{!_A} A$ denotes the unique function of its type: there is a maximum loss of injectivity and all data become synonyms!

Hashing as a Galois connection. As powerset-valued functions, hash tables are ordered by the lifting of the subset ordering $\mathcal{P}(A) \xleftarrow{\leq} \mathcal{P}(A)$ defined by $\leq = \in \setminus \in$, recall (3.14).

That the construction of hash tables is monotonic can be shown using the relational calculus. First we expand $\dot{\leq}$ (2.10):

$$
\begin{array}{ll}
& t \;\; \dot{\leq} \;\; t' \\
\equiv & \quad \{ \text{ pointwise ordering lifted to functions (2.10) } \} \\
& t \;\; \subseteq \;\; \leq \cdot\, t' \\
\equiv & \quad \{ \text{ definition of the subset ordering (3.14) } \} \\
& t \;\; \subseteq \;\; (\in \setminus \in) \cdot t' \\
\equiv & \quad \{ \text{ law } (R \setminus S) \cdot f = R \setminus (S \cdot f) \text{ [29], since } t' \text{ is a function } \} \\
& t \;\; \subseteq \;\; \in \setminus (\in \cdot\, t') \\
\equiv & \quad \{ \; (\in\cdot) \text{ is lower adjoint of } (\in\setminus) \text{, recall (2.15) } \} \\
& \in \cdot\, t \;\; \subseteq \;\; \in \cdot\, t'
\end{array}
\tag{4.5}
$$

Then we reason:

$$
\begin{array}{ll}
& (\Theta_h)S \;\; \dot{\leq} \;\; (\Theta_h)R \\
\equiv & \quad \{ \text{ by (4.5) } \} \\
& \in \cdot\, (\Theta_h)S \;\; \subseteq \;\; \in \cdot\, (\Theta_h)R \\
\equiv & \quad \{ \text{ cancellation (4.4) } \} \\
& S \cdot h^\circ \;\; \subseteq \;\; R \cdot h^\circ \\
\Leftarrow & \quad \{ \; (\cdot h^\circ) \text{ is monotone, cf. lower-adjoints in (2.15) } \} \\
& S \;\; \subseteq \;\; R
\end{array}
$$

So, the smallest of all hash-tables is that associated with the empty relation $\bot$, that is $\Lambda\bot$, which is constant function $t = \underline{\emptyset}$, and the largest one is $t = \Lambda h^\circ$, the hash-transpose of id_A. In set-theoretic terms, this is A itself, the "largest" set of data of type A.

That the hash-transpose is not an isomorphism is intuitive: not every function t mapping B to $\mathcal{P}(A)$ will be a hash-table, because it may fail to place data in the correct bucket. Anyway, it is always possible to "filter" the wrongly placed synonyms from t yielding the "largest" (correct) hash table t' it contains,

$$t' = t \mathbin{\dot\cap} \Lambda(h^\circ)$$

where, using *vector notation* [13], $f \mathbin{\dot\cap} g$ is the lifting of $\cap$ to powerset-valued functions,

$$(f \mathbin{\dot\cap} g)b = (f\ b) \cap (g\ b)$$

for all b.

In order to recover all data from such filtered t' we evaluate

$$\rho\,(\in \cdot\, t')$$

where $\rho\, R$ (2.15) means *img* $R \cap id$. Altogether, we may define a function on powerset valued functions $\Xi_h t = \rho\,(\in \cdot\,(t \mathbin{\dot\cap} \Lambda(h^\circ)))$ which extracts the coreflexive relation associated with all data correctly placed in t via hash function h. By reconverting $\Xi_h t$ into a hash-table again one will get a table smaller than t:

$$\Theta_h(\Xi_h t) \quad \dot\leq \quad t \tag{4.6}$$

Another fact one can prove is the "perfect" cancellation on the other side:

$$\Xi_h(\Theta_h S) \quad = \quad S \tag{4.7}$$

These two cancellations, together with the monotonicity of the hash transpose Θ_h and that of Ξ_h (this is monotone because it only involves monotonic combinators) are enough, by Theorem 5.24 in [1], to establish *perfect* Galois connection

$$\Theta_h S \mathrel{\dot\leq} t \quad \equiv \quad S \subseteq \Xi_h\, t$$

cf. diagram

$$\begin{tikzcd} \{S \mid S \subseteq id_A\} \arrow[loop above, "\subseteq"] \arrow[r, shift left, "\Theta_h"] & (\mathcal{P}(A))^B \arrow[loop above, "\dot\leq"] \arrow[l, shift left, "\Xi_h"] \end{tikzcd} \tag{4.8}$$

Being a lower adjoint, the hash-transpose will distribute over union, $\Theta_h(R \cup S) = (\Theta_h R) \mathbin{\dot\cup} (\Theta_h S)$ (so hash-table construction is compositional) and enjoy other properties known of Galois connections.

From (4.7) we infer that Θ_h (resp. Ξ_h) is injective (resp. surjective) and so can be regarded as a data *representation* (resp. *abstraction*) in the terminology of Fig. 2.1, whereby typical "database" operations such as *insert*, *find*, and *remove* (specified on top of the powerset algebra) can be implemented by calculation [107].

As a way of expressing that the left hand side of (4.8) is smaller than its right hand side we write

$$
\begin{tikzcd}
\{S \mid S \subseteq A\} \arrow[rr, bend left, "\Theta_h"] \arrow[rr, phantom, "\leq"] & & (\mathcal{P}(A))^B \arrow[ll, bend left, "\Xi_h"]
\end{tikzcd}
\tag{4.9}
$$

The meaning of the $\leq$-relation is discussed in what follows.

4.3 An Introduction to SETS

The SETS data refinement calculus [110, 112, 109] is based on laws such as (4.9) which are known as $\leq$-facts, $\leq$-rules or data refinement rules. Another example of such rules is

$$
\begin{tikzcd}
\mathcal{P}_f\, A \arrow[r, bend left] \arrow[r, phantom, "\leq"] & A^\star \arrow[l, bend left, "elems"]
\end{tikzcd}
$$

which expresses the fact that finite sets are implementable by finite seqences. There are several alternative meanings for this $\leq$-rule:

- finite sets are "implemented" by finite lists
- $A^\star$ is able to "represent" $\mathcal{P}_f\, A$
- $A^\star$ is "abstracted" by $\mathcal{P}_f\, A$
- $A^\star$ is a refinement ("refines") $\mathcal{P}_f\, A$

Function $elems$, which extracts the set of all elements of a finite list, is one of the *witnesses* of this data refinement rule. In general, the fact that A *is implemented by* B, as *witnessed* by abstraction/representation pair f, r, and recorded by diagram

$$
\begin{tikzcd}
A \arrow[r, bend left, "r"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "f"]
\end{tikzcd}
$$

means that $f \cdot r = id$, that is to say

- **representation** r is injective
- **abstraction** f is surjective

The $\leq$ sign captures the fact that the cardinality of A is smaller than that of B.

In practice, we need a still more general definition, because f is not always totally defined and r is not always a function. For instance, one has $i_1^\circ \cdot i_1 = id$, cf.

$$
\begin{tikzcd}
A \arrow[r, bend left, "i_1"] \arrow[r, phantom, "\leq"] & A + 1 \arrow[l, bend left, "i_1^\circ"]
\end{tikzcd}
$$

expressing the fact that every element of datatype A can be represented by a "*pointer*". Clearly,

- $r = i_1$ is injective, but
- its converse i_1° is *partial* (not entirely defined).

On the other hand, representation r need not be a function, as can be seen going back to

$$\begin{tikzcd} \mathcal{P}_f\, A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & A^\star \arrow[l, bend left, "elems"] \end{tikzcd}$$

and taking $R = elems^\circ$. This will be perfectly acceptable representation since

$$elems \cdot elems^\circ = id$$

holds ($elems$ is a surjection). We are lead to the following principle of data abstraction:

Definition 4.1 (Principle of Data Abstraction) *Two datatypes A and B are such A is more abstract than B, written*

$$\begin{tikzcd} A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "F"] \end{tikzcd}$$

iff

- $A \xleftarrow{F} B$ *is a* ***surjective*** + ***simple*** abstraction *relation*
- R *is* ***entire*** *and* $R \subseteq F^\circ$ *— it is said to be a* representation *for* F.

Condition $R \subseteq F^\circ$ (equivalent to $R^\circ \subseteq F$) expresses the fact that R, F are connected *to each other. We often say that R, F are a connected representation/abstraction pair.*

The fact that R is **injective** follows from $R \subseteq F^\circ$ (since converse of simple is injective and at most injective is injective). In summary:

(4.10)

ker $R = id$	entire R $\wedge$ injective R	representation R
img $F = id$	surjective F $\wedge$ simple F	abstraction F

It follows that R is a **right-inverse** of F, that is

$$F \cdot R \quad = \quad id$$

still holds. This is proved below by circular inclusion

$$F \cdot R \subseteq id \subseteq F \cdot R$$

as follows:

$$
\begin{array}{cl}
 & F \cdot R \subseteq id \ \wedge \ id \subseteq F \cdot R \\
\equiv & \qquad \{ \ \mathsf{img}\, F = id \text{ and } \mathsf{ker}\, R = id \text{ (4.10)} \} \\
 & F \cdot R \subseteq F \cdot F^{\circ} \ \wedge \ R^{\circ} \cdot R \subseteq F \cdot R \\
\equiv & \qquad \{ \ \text{converse of right conjunct } \} \\
 & F \cdot R \subseteq F \cdot F^{\circ} \ \wedge \ R^{\circ} \cdot R \subseteq R^{\circ} \cdot F^{\circ} \\
\Leftarrow & \qquad \{ \ (F\cdot) \text{ and } (R^{\circ}\cdot) \text{ are monotonic } \} \\
 & R \subseteq F^{\circ} \ \wedge \ R \subseteq F^{\circ} \\
\equiv & \qquad \{ \ R \subseteq F^{\circ} \text{ is assumed } \} \\
 & \textsc{True}
\end{array}
$$

Theorem 4.1 ($\leq$ is a preorder) Proof:

Reflexivity is obvious to express:

$$
\begin{tikzcd}
A \arrow[r, bend left, "id"] \arrow[r, phantom, "\leq"] & A \arrow[l, bend left, "id"]
\end{tikzcd}
$$

The proof of transitivity

$$
\begin{tikzcd}
A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "F"]
\end{tikzcd}
\ \wedge \
\begin{tikzcd}
B \arrow[r, bend left, "S"] \arrow[r, phantom, "\leq"] & C \arrow[l, bend left, "G"]
\end{tikzcd}
\ \Rightarrow \
\begin{tikzcd}
A \arrow[r, bend left, "S \cdot R"] \arrow[r, phantom, "\leq"] & C \arrow[l, bend left, "F \cdot G"]
\end{tikzcd}
$$

is carried out in two steps:

a) Composition preserves simplicity and surjectiveness:

$$
\begin{array}{cl}
 & \mathsf{img}\,(F \cdot G) = id \\
\equiv & \qquad \{ \textit{expand } img \ \textit{; converses} \} \\
 & F \cdot (\mathsf{img}\, G) \cdot F^{\circ} = id \\
\equiv & \qquad \{ \ G \textit{ is simple and surjective} \} \\
 & \mathsf{img}\, F = id \\
\equiv & \qquad \{ \ F \textit{ is simple and surjective} \} \\
 & id = id
\end{array}
$$

b) $S \cdot R$ *and* $F \cdot G$ *are connected:*

$$
\begin{array}{ll}
& S \cdot R \subseteq (F \cdot G)^{\circ} \\
\equiv & \qquad \{\textit{contravariance}\} \\
& S \cdot R \subseteq G^{\circ} \cdot F^{\circ} \\
\Leftarrow & \qquad \{\textit{monotonicity}\} \\
& S \subseteq G^{\circ} \wedge R \subseteq F^{\circ} \\
\equiv & \qquad \{\textit{by hypothesis}\} \\
& True
\end{array}
$$

Theorem 4.2 (Data refinement is structural) *For* F *an arbitrary relator,*

$$
A \underset{F}{\overset{R}{\rightleftarrows}} B \ (\leq) \quad \Rightarrow \quad \mathsf{F}\,A \underset{\mathsf{F}\,f}{\overset{\mathsf{F}\,R}{\rightleftarrows}} \mathsf{F}\,B \ (\leq) \tag{4.11}
$$

Proof:

The proof of (4.11) is carried out in three steps:

a) $(F\ f)$ *is an* abstraction*:*

$$
\begin{array}{ll}
& \mathit{img}\,(\mathsf{F}\,f) \\
= & \qquad \{\ \textit{image definition ; relators commute with converse}\ \} \\
& (\mathsf{F}(f^{\circ})) \cdot (\mathsf{F}\,f) \\
= & \qquad \{\ \textit{relators commute with composition}\ \} \\
& \mathsf{F}(f^{\circ} \cdot f) \\
= & \qquad \{\ f \textit{ is an abstraction}\ \} \\
& \mathsf{F}\,id \\
= & \qquad \{\textit{relators commute with } id\} \\
& id
\end{array}
$$

b) $F\ R$ *is a* representation:

$$
\begin{array}{ll}
& \mathsf{F}\,R \ \subseteq \ (\mathsf{F}\,f)^{\circ} \\
\equiv & \qquad \{\ \textit{relators commute with converse}\ \} \\
& \mathsf{F}\,R \ \subseteq \ \mathsf{F}(f^{\circ}) \\
\Leftarrow & \qquad \{\ \textit{relators are monotone}\ \} \\
& R \ \subseteq \ f^{\circ} \\
\equiv & \qquad \{\ R \textit{ is a representation for } f\ \} \\
& \textsc{True}
\end{array}
$$

c) $\mathsf{F}\ f$ *and* $\mathsf{F}\ R$ *are connected:*

$$
\begin{array}{ll}
 & True \\
\equiv & \qquad \{ \textit{ by hypothesis} \} \\
 & R \subseteq F^{\circ} \\
\Rightarrow & \qquad \{ \textit{ relator} \} \\
 & \mathsf{F}\ R \subseteq \mathsf{F}\ F^{\circ} \\
\Rightarrow & \qquad \{ \textit{ relator} \} \\
 & \mathsf{F}\ R \subseteq (\mathsf{F}\ F)^{\circ}
\end{array}
$$

4.4 Catalogue of $\leq$-rules

Now it's time to present a glimpse of the calculus [112, 115, 111], concerning the refinement of finite simple relations by decomposition. These laws will be proven in sections to follow. The first law removes linear multiplicative structure from the range of a simple relation.

$$
\begin{tikzcd}
A \rightharpoonup B \times C \arrow[rr, bend left, "unjoin"] & \leq & (A \rightharpoonup B) \times (A \rightharpoonup C) \arrow[ll, bend left, "join = \langle \_, \_ \rangle"]
\end{tikzcd}
\tag{4.12}
$$

where

$$
\begin{array}{rcl}
\langle R, S \rangle & \stackrel{\text{def}}{=} & (\pi_1^{\circ} \cdot R) \cap (\pi_2^{\circ} \cdot S) \\
unjoin & \stackrel{\text{def}}{=} & \langle id \rightharpoonup \pi_1, id \rightharpoonup \pi_2 \rangle
\end{array}
$$

where, for injective f,

$$
(f \rightharpoonup g)\ R \stackrel{\text{def}}{=} g \cdot R \cdot f^{\circ}
$$

The following law removes additive structure instead:

$$
\begin{tikzcd}
A \rightharpoonup (B + C) \arrow[rr, bend left, "uncojoin"] & \leq & (A \rightharpoonup B) \times (A \rightharpoonup C) \arrow[ll, bend left, "cojoin"]
\end{tikzcd}
\tag{4.13}
$$

where

$$
\begin{array}{rcl}
uncojoin & = & \langle (i_1^{\circ} \cdot), (i_2^{\circ} \cdot) \rangle \\
cojoin & = & \cup \cdot ((i_1 \cdot) \times (i_2 \cdot))
\end{array}
$$

The following law is an elaboration of the first one, where one has to remove a nested structure.

$$\begin{tikzcd} A \rightharpoonup (D \times (B \rightharpoonup C)) \arrow[r, bend left, "\Delta_n"] \arrow[r, phantom, "\leq"] & (A \rightharpoonup D) \times ((A \times B) \rightharpoonup C) \arrow[l, bend left, "\bowtie_n"] \end{tikzcd} \tag{4.14}$$

where

$$R \bowtie_n S = \langle R, \overline{S} \rangle$$

$$\Delta_n\, R = (\pi_1 \cdot R, usc(\pi_2 \cdot R))$$

4.4.1 Isomorphisms

Isomorphisms are special cases of ≤-facts. They form a sub-calculus of their own.

We start with a relational version of *currying*:

Lemma 4.1 (Relational currying) *The following law caracterizes a version of currying/uncurrying in the context of simple relations.*

$$\begin{tikzcd} B \times C \rightharpoonup A \arrow[r, bend left, "scurry"] \arrow[r, phantom, "\cong"] & (C \rightharpoonup A)^B \arrow[l, bend left, "usc"] \end{tikzcd} \tag{4.15}$$

and which is easy to justify on the grounds of (3.16):

$$\begin{array}{cl} & (B \times C) \rightharpoonup A \\ \equiv & \qquad \{\ (3.16)\} \\ & (A+1)^{B \times C} \\ \equiv & \qquad \{\ (un)currying\} \\ & ((A+1)^C)^B \\ \equiv & \qquad \{\ (3.16)\ and\ exponential\} \\ & (C \rightharpoonup A)^B \end{array}$$

The following notation will be introduced for isomorphism $scurry$: $scurry\ S = \overline{S}$.

Finally an isomorphism law, removing additive structure from the domain of a simple relation:

Lemma 4.2

$$\begin{tikzcd} B + C \rightharpoonup A \arrow[r, bend left, "unpeither"] \arrow[r, phantom, "\cong"] & B \rightharpoonup A \times C \rightharpoonup A \arrow[l, bend left, "peither"] \end{tikzcd}$$

where

$$peither = [\_, \_]$$

and $unpeither = peither^{\circ}$.

The following law has been already given as (3.16):

Lemma 4.3 (Transposition)

$$\begin{tikzcd} (B+1)^A \arrow[r, bend left, "(i_1^{\circ} \cdot)"] \arrow[r, phantom, "\cong"] & A \rightharpoonup B \arrow[l, bend left, "\Gamma"] \end{tikzcd} \tag{4.16}$$

We shall present 3 isomorphisms in the next 3 lemmas whose proofs are given in a constructive style.

Lemma 4.4

$$\begin{tikzcd} 2^A \arrow[r, bend left, "r"] \arrow[r, phantom, "\cong"] & (1+1)^A \arrow[l, bend left, "a"] \end{tikzcd}$$

Proof: It can be easily shown that $[\underline{1}, \underline{2}]$ *in*

$$\begin{tikzcd} 2 \arrow[r, bend left] \arrow[r, phantom, "\cong"] & (1+1) \arrow[l, bend left, "{[\underline{1}, \underline{2}]}"] \end{tikzcd}$$

is a bijection. Thus so is $[\underline{1}, \underline{2}]^A$*, since functors preseve bijections. Thus we have*

$$\begin{tikzcd} 2^A \arrow[r, bend left] \arrow[r, phantom, "\cong"] & (1+1)^A \arrow[l, bend left, "{[\underline{1}, \underline{2}]^A}"] \end{tikzcd}$$

Lemma 4.5 (Sets are fragments of "bang")

$$\begin{tikzcd} A \rightharpoonup 1 \arrow[r, bend left, "r"] \arrow[r, phantom, "\cong"] & \mathcal{P}(A) \arrow[l, bend left, "f"] \end{tikzcd}$$

Proof: *This folllows from (4.16) by letting* $B = 1$*, since* $1 + 1 \cong 2$ *and* $\mathcal{P}(A) \cong 2^A$.

In SETS, the following isomorphism is called $set2fm$, a function which converts sets to finite mappings. In pointfree notation, sets are represented by coreflexives and $set2fm = (! \cdot)$.

Lemma 4.6

$$
\begin{tikzcd}
2^A \arrow[r, bend left] \arrow[r, phantom, "\cong"] & A \rightharpoonup 1 \arrow[l, bend left]
\end{tikzcd}
$$

is an isomorphism too. It can be derived from other isomorphisms.

The derivation is,

$$
\begin{array}{ll}
 & 2^A \\
\equiv & \quad \{ \begin{cases} a_1 = cfs \stackrel{\text{def}}{=} \lambda y : A.y \in s & \} \\ r_1 = zf\phi \stackrel{\text{def}}{=} \{x \in A | \phi x\} \end{cases} \\
 & P\ A \\
\equiv & \quad \{ \begin{cases} a_2 = dom & \} \\ r_2 = set2fm \stackrel{\text{def}}{=} \lambda s \cdot \{x \mapsto NIL) | x \in s\} \end{cases} \\
 & A \rightharpoonup 1
\end{array}
$$

There is a version of r_2 in poinfree notation, which reads

$$r_2 = set2fm = (!\cdot)$$

The overall isomorphisms are,

$$
\begin{array}{rcl}
a & = & a_1 \cdot a_2 \\
 & = & cf \cdot dom
\end{array}
$$

and

$$
\begin{array}{rcl}
r & = & r_2 \cdot r_1 \\
 & = & set2fm \cdot zf
\end{array}
$$

Next, we shall calculate a simplified version of both isomorphisms. Begining with a, in point-wise notation,

$$
\begin{array}{ll}
 & cf(dom\ \sigma) \\
= & \quad \{ \textit{definition of cf} \} \\
 & \lambda x : A.x \in (dom\ \sigma)
\end{array}
$$

Now we consider r, also in pointwise notation

$$
\begin{array}{cl}
 & \lambda s.\{(y \mapsto NIL) | y \in s\}(zf\phi) \\
= & \qquad \{\ \beta\text{-}\textit{reduction}\} \\
 & \{(y \mapsto NIL) | y \in (zf\phi)\} \\
= & \qquad \{\ \textit{definition of zf}\} \\
 & \{(y \mapsto NIL) | y \in \{x \in A | \phi\, x\}\} \\
= & \qquad \{\ \textit{eliminating a membership relation}\ \} \\
 & \{(y \mapsto NIL) | y = x \wedge x \in A \wedge \phi\, x\} \\
= & \qquad \{\ \textit{replace x by y}\} \\
 & \{(y \mapsto NIL) | y \in A \wedge \phi\, y\}
\end{array}
$$

The next 2 isomorphisms relate a data type of mappings, either total or partial, with values and "pointers", respectively.

Lemma 4.7

$$
\begin{tikzcd}
A^1 \arrow[r, bend left] \arrow[r, phantom, "\cong"] & A \arrow[l, bend left]
\end{tikzcd}
$$

is a basic isomorphism. Proof: We shall present its witnesses,

$$
\begin{tikzcd}
A^1 \arrow[r, bend left, "f2v"] \arrow[r, phantom, "\cong"] & A \arrow[l, bend left, "v2f"]
\end{tikzcd}
\tag{4.17}
$$

where

$$
\begin{array}{rcl}
v2f(a) & \stackrel{\text{def}}{=} & \{\underline{a}\} \\
f2v(f) & = & f(NIL)
\end{array}
$$

It's easy to realize that the two witnesses are each other inverses.

Lemma 4.8

$$
\begin{tikzcd}
1 \rightharpoonup A \arrow[r, bend left] \arrow[r, phantom, "\cong"] & A + 1 \arrow[l, bend left]
\end{tikzcd}
$$

is the other isomorphism. The derivation is,

$$
\begin{array}{cl}
 & 1 \rightharpoonup A \\
\equiv & \qquad \{\ \textit{(4.16)}\ \} \\
 & (A+1)^1 \\
 & \\
\equiv & \qquad \{\ \textit{(4.17)}\ \} \\
 & A + 1
\end{array}
$$

The overall isomorphisms are,

$$\begin{aligned} a &= a_1 \cdot a_2 \\ &= \Gamma^\circ \cdot v2f \end{aligned}$$

and

$$\begin{aligned} r &= r_2 \cdot r_1 \\ &= f2v \cdot \Gamma \end{aligned}$$

which convert to points as follows:

$$\begin{array}{ll} & a\ x \\ = & \qquad \{\ \} \\ & \Gamma^\circ(v2fx) \\ = & \qquad \{\ \textit{definition of v2f and } \Gamma^\circ = (i_1^\circ \cdot)\ \textit{(3.8)}\} \\ & i_1^\circ \cdot \underline{x} \end{array}$$

and

$$\begin{array}{ll} & r\ \sigma \\ = & \qquad \{\ \} \\ & f2v(\Gamma(\sigma)) \\ = & \qquad \{\ \textit{definition of f2v}\} \\ & \Gamma(\sigma)(NIL) \end{array}$$

4.5 SETS Laws as Galois Connections

Our introdutory example in section 4.2 lead to the conclusion that the hashing effect can be explaned by a Galois connection. It so happens with a significant set of $\leq$-rules. So we now start a more thorough study of $\leq$-rules based on Galois connections. We begin with rule (4.12), which is supported by a universal property:

Proposition 4.1 *(Relational Split) The following universal property*

$$X \subseteq \langle R, S \rangle \quad \equiv \quad \pi_1 \cdot X \subseteq R \wedge \pi_2 \cdot X \subseteq S \tag{4.18}$$

captures the meaning of relational split *as a Galois connection,*

$$\begin{tikzcd} (\ A \rightharpoonup B \times C, \subseteq) \arrow[r, phantom, "\leq"] \arrow[r, bend left, "r"] & ((A \rightharpoonup B) \times (A \rightharpoonup C), \subseteq \times \subseteq) \arrow[l, bend left, "f"] \end{tikzcd} \tag{4.19}$$

where

$$f = \langle \_, \_ \rangle$$

and

$$r = ((\pi_1 \cdot) \times (\pi_2 \cdot)) \cdot \Delta$$

such that $\Delta\ X = (X, X)$.

Proposition 4.2 *f and r in law (4.19) are connected ($r \subseteq f^\circ$) because the connection is perfect on the lefthand side, that is,*

$$\langle \pi_1 \cdot S, \pi_2 \cdot S \rangle = S$$

Proof:

We shall do the proof for simple relations. The complete proof will come in the section dedicated to relations.

$$\begin{array}{cl} & \textsc{True} \\ \equiv & \qquad \{ \textit{reflexivity} \} \\ & S = S \\ \equiv & \qquad \{ \textit{reflexion} \} \\ & \langle \pi_1, \pi_2 \rangle \cdot S = S \\ \equiv & \qquad \{ \textit{fusion for simple relations (B.10)} \} \\ & \langle \pi_1 \cdot S, \pi_2 \cdot S \rangle = S \end{array}$$

The following Galois connection is an extension of isomorphism (4.15):

$$\begin{tikzcd} B \times C \rightharpoonup A \arrow[rr, bend left, "(\overline{\_})"] \arrow[rr, phantom, "\leq"] & & B \rightharpoonup (C \rightharpoonup A) \arrow[ll, bend left, "usc"] \end{tikzcd} \quad \text{if} \quad usc\, M \subseteq S \;\equiv\; M \mathrel{\dot{\subseteq}} \overline{S} \tag{4.20}$$

where we extend functional lifting (2.10) of a partial order $\leq$ to simple relations, as follows,

$$M \mathrel{\dot{\leq}} N \quad \stackrel{\text{def}}{=} \quad M \subseteq \leq \cdot N \tag{4.21}$$

and where

$$usc\, M \quad \stackrel{\text{def}}{=} \quad \langle \bigcap\, S \;:\; M \mathrel{\dot{\subseteq}} \overline{S} :\; S \rangle \tag{4.22}$$

Since upper adjoint $(\overline{\_})$ is monotonic (it is an isomorphism), definition (4.22) ensures Galois connection (4.20) by Theorem 5.32 of [1].

This Galois connection is perfect, as the following lemma shows.

Lemma 4.9 (GC (4.20) is perfect on the lower side)

$$usc(\overline{X}) = X$$

Proof: *We know that*

$$usc(\overline{X}) \subseteq X$$

holds by right cancellation. We are left with proving $X \subseteq usc(\overline{X})$*:*

$$
\begin{array}{ll}
 & X \subseteq usc(\overline{X}) \\
\equiv & \qquad \{ \textit{ by definition} \} \\
 & X \subseteq \langle \bigcap\, S \ :\ \overline{X} \mathrel{\dot{\subseteq}} \overline{S} :\ S \rangle \\
\equiv & \qquad \{ \textit{ universal property of meet} \} \\
 & \langle \forall\, S \ :\ \overline{X} \mathrel{\dot{\subseteq}} \overline{S} :\ X \subseteq S \rangle \\
\equiv & \qquad \{ \textit{ } \forall\textit{-trading [12]} \} \\
 & \langle \forall\, S \ ::\ \overline{X} \mathrel{\dot{\subseteq}} \overline{S} \Rightarrow X \subseteq S \rangle \\
\equiv & \qquad \{ \textit{ } \overline{X}, \overline{S} \textit{ are functions; isomorphism (4.15) } \} \\
 & \textsc{True}
\end{array}
$$

Finally, we justify law (4.14) in the form of yet another perfect Galois connection. We will need the following standard result [1]:

Definition 4.2 *Galois connection (f,g) is a Galois connection iff the following two clauses hold*

- f *and* g *are monotonic,*
- $x \sqsubseteq (g \cdot f)\ x$ *and* $(f \cdot g)\ y \sqsubseteq y$

In the proof of the next proposition we will need the following equations relating projections and split,

$$\pi_1 \cdot \langle R, S \rangle = R \cdot \delta\, S \tag{4.23}$$
$$\pi_2 \cdot \langle R, S \rangle = S \cdot \delta\, R \tag{4.24}$$

Proposition 4.3 *Nested join is an adjoint of GC:*

$$
\begin{tikzcd}
(A \rightharpoonup (D \times (B \rightharpoonup C)), \dot{\sqsubseteq}) \arrow[r, bend left, "\Delta_n"] \arrow[r, phantom, "\leq"] & ((A \rightharpoonup D) \times ((A \times B) \rightharpoonup C), \subseteq \times \subseteq) \arrow[l, bend left, "\bowtie_n"]
\end{tikzcd}
\tag{4.25}
$$

where

$$\dot{\sqsubseteq} = id \times \subseteq$$

and

$$R \bowtie_n S = \langle R, \overline{S} \rangle$$

and

$$\Delta_n\, R = (\pi_1 \cdot R, usc(\pi_2 \cdot R))$$

Proof: *We start by noting that both Δ_n and $\ltimes_n$ are monotonic since they are compositions of monotonic operators. According to Definition 4.2, we are left with the proof that cancellations*

$$\begin{array}{rcl} (\Delta_n \cdot \ltimes_n)(R,S) & \subseteq \times \subseteq & (R,S) \\ R & \dot{\sqsubseteq} & (\ltimes_n \cdot \Delta_n)\ R \end{array}$$

hold. Concerning the former, we calculate:

$$\begin{array}{cl} & \Delta_n(\ltimes_n\ (R,S)) \subseteq \times \subseteq (R,S) \\ \equiv & \quad \{ \textit{definition of } \Delta_n \textit{ and of } \ltimes_n \} \\ & (\pi_1 \cdot \langle R, \overline{S} \rangle, usc(\pi_2 \cdot \langle R, \overline{S} \rangle)) \subseteq \times \subseteq (R,S) \\ \equiv & \quad \{ \textit{projections of splits (4.24)} \} \\ & (R \cdot \delta\, \overline{S}, usc(\overline{S} \cdot \delta\, R)) \subseteq \times \subseteq (R,S) \\ \equiv & \quad \{ \overline{S} \textit{ is a function} \} \\ & usc(\overline{S} \cdot \delta\, R) \subseteq S \\ \equiv & \quad \{ usc\, \overline{S} = S, \textit{GC (4.20)} \} \\ & usc(\overline{S} \cdot \delta\, R) \subseteq usc\, \overline{S} \\ \Leftarrow & \quad \{ usc \textit{ is monotonic} \} \\ & \overline{S} \cdot \delta\, R \subseteq \overline{S} \\ \equiv & \quad \{ \textit{coreflexive } \delta\, R \} \\ & True \end{array}$$

Now the other cancellation:

$$\begin{array}{cl} & R \mathrel{\dot{\sqsubseteq}} (\ltimes_n \cdot \Delta_n)\ R \\ \equiv & \quad \{ \textit{reflexion, fusion; definitions} \} \\ & \langle \pi_1 \cdot R, \pi_2 \cdot R \rangle \mathrel{\dot{\sqsubseteq}} \langle \pi_1 \cdot R, \overline{usc(\pi_2 \cdot R)} \rangle \\ \equiv & \quad \{ \textit{expand } \dot{\sqsubseteq} \textit{ and absorption} \} \\ & \langle \pi_1 \cdot R, \pi_2 \cdot R \rangle \subseteq \langle \pi_1 \cdot R, (\subseteq) \cdot \overline{usc(\pi_2 \cdot R)} \rangle \\ \Leftarrow & \quad \{ \textit{relational split is monotonic} \} \\ & \pi_1 \cdot R \subseteq \pi_1 \cdot R \wedge \pi_2 \cdot R \mathrel{\dot{\subseteq}} \overline{usc(\pi_2 \cdot R)} \\ \equiv & \quad \{ \textit{GC (4.20)} \} \\ & \textsc{True} \end{array}$$

To complete the proof of $\leq$-rule (4.25) we need to ensure invertibility:

Lemma 4.10 (GC (4.25) is perfect)

$$(\ltimes_n \cdot \Delta_n)\ R = R$$

Proof:

$$
\begin{array}{ll}
& (\bowtie_n \cdot \Delta_n)\; R = R \\
\equiv & \quad \{ \text{ by definition; reflexion and fusion, cf. (B.10) } \} \\
& \langle \pi_1 \cdot R, \overline{usc(\pi_2 \cdot R)} \rangle = \langle \pi_1 \cdot R, \pi_2 \cdot R \rangle \\
\equiv & \quad \{ \text{ see (4.26) below } \} \\
& \langle \pi_1 \cdot R, \overline{usc(\pi_2 \cdot R)} \cdot \delta\, R \rangle = \langle \pi_1 \cdot R, \pi_2 \cdot R \rangle \\
\equiv & \quad \{ \text{ see (4.27) below } \} \\
& \mathrm{TRUE}
\end{array}
$$

The proofs of the two laws assumed in the calculation above

$$
\begin{array}{rcl}
\langle R, S \rangle & = & \langle R \cdot \delta\, S, S \cdot \delta\, R \rangle \qquad (4.26) \\
\overline{uscX} \cdot \delta\, X & = & X \qquad (4.27)
\end{array}
$$

can be found in appendix B.5.

There is no possible Galois connected law for $cojoin$, since this operator is partial (thus not a function). We shall study a corresponding law for the data type of relations, to be studied in section 4.7.

4.6 Calculation of the Invariants induced by the SETS' inequations

We are now interested in calculating the invariant induced by a abs/rep pair f, r which, by definition, is the range of r. The set of all c, such that $(r \cdot f)c = c$ is the range of r, as we justify next, and therefore the intended invariant ϕ can be defined as $\phi\; c \stackrel{\text{def}}{=} (r \cdot f)\; c = c$.

We begin with the following lemma whose proof is taken from [12].

Lemma 4.11 *If f and r are Galois connected then*

$$x \in \rho\, r \equiv r(f\; x) = x$$

Proof:

$$
\begin{array}{ll}
 & x \in \rho\, r \\
\equiv & \quad \{ \textit{definition of } \rho\, r \} \\
 & \langle \exists\, y \;::\; x = r\ y \rangle \\
\Rightarrow & \quad \{ \textit{semi-inverse: } r \cdot f \cdot r = r \} \\
 & r(f\ x) = x \\
\Rightarrow & \quad \{ y := f\ x \} \\
 & \langle \exists\, y \;::\; x = r\ y \rangle \\
\equiv & \quad \{ \textit{definition of } \rho\, r \} \\
 & x \in \rho\, r
\end{array}
$$

Now we transform the above equivalence to the binary pointfree relational calculus

$$r \cdot r^{\circ} = r \cdot f \cap id \tag{4.28}$$

recalling that, in the case of functions, range and image coincide:

$$
\begin{array}{ll}
 & x \in \rho\, r \equiv r(f\ x) = x \\
\equiv & \quad \{ \text{introducing } rng \} \\
 & x(\rho\, r)x \equiv r(f\ x) = x \\
\equiv & \quad \{ \text{going pointfree} \} \\
 & \rho\, r = r \cdot f \cap id \\
\equiv & \quad \{ \text{definition of } \rho \text{ for simple relations} \} \\
 & r \cdot r^{\circ} = r \cdot f \cap id
\end{array}
$$

Fact (4.28) is of help in the context of data refinement where the pair f and r may be taken as the functional abstraction/representation relations. It relates the interpretation of the concrete invariant induced by a data refinement with the pointfree invertibility condition, which are equated.

Let us apply this result to law (4.12), which we restate in terms of a Galois connection:

$$
\begin{tikzcd}
A \rightharpoonup (B \times C) \arrow[rr, phantom, "\leq"] \arrow[rr, bend left, "r = \langle \pi_1 \rightharpoonup id, \pi_2 \rightharpoonup id \rangle"] & & (A \rightharpoonup B) \times (A \rightharpoonup C) \arrow[ll, bend left, "f = \langle \_, \_ \rangle"]
\end{tikzcd}
\tag{4.29}
$$

The mentioned condition will be, in this case,

$$\langle \pi_1 \rightharpoonup id, \pi_2 \rightharpoonup id \rangle(\langle R, S \rangle) \quad = \quad (R, S)$$

that is

$$\begin{array}{ll} & (R,S) \quad = \quad (\pi_1 \cdot \langle R,S \rangle, \pi_2 \cdot \langle R,S \rangle) \\ \equiv & \qquad \{ \text{structural equality} \} \\ & R = \pi_1 \cdot \langle R,S \rangle \quad \wedge \quad S = \pi_2 \cdot \langle R,S \rangle \\ \equiv & \qquad \{ \text{ relational split cancellation (4.30) } \} \\ & \delta\, R = \delta\, S \end{array}$$

So we have to prove

$$R = \pi_1 \cdot \langle R,S \rangle \quad \wedge \quad S = \pi_2 \cdot \langle R,S \rangle \quad \equiv \quad \delta\, R = \delta\, S \tag{4.30}$$

We begin (for the case of R, the case of S is similar):

$$\begin{array}{ll} & R = \pi_1 \cdot \langle R,S \rangle \\ \equiv & \qquad \{ \ \langle R,S \rangle \subseteq \pi_1^\circ \cdot R \text{ holds (2.24)} \} \\ & R \subseteq \pi_1 \cdot \langle R,S \rangle \\ \equiv & \qquad \{ \text{ definition (2.24) and (4.31) below } \} \\ & R \subseteq \pi_1 \cdot \pi_1^\circ \cdot R \cap \pi_1 \cdot \pi_2^\circ \cdot S \\ \equiv & \qquad \{ \ \pi_1 \text{ is a surjection and } \pi_1 \cdot \pi_2^\circ = \top \ \} \\ & R \subseteq R \cap \top \cdot S \\ \equiv & \qquad \{ \text{ meet-universal and } \top = \mathit{ker}\,! \ \} \\ & ! \cdot R \subseteq \, ! \cdot S \\ \equiv & \qquad \{ \text{ conditions (2.17) } \} \\ & \delta\, R \subseteq \delta\, S \end{array}$$

The fact assumed above is

$$(\pi_1 \cdot \langle R,S \rangle =) \pi_1 \cdot (\pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S) = \pi_1 \cdot \pi_1^\circ \cdot R \cap \pi_1 \cdot \pi_2^\circ \cdot S \tag{4.31}$$

For a proof of (4.31) see

$$R \cdot (S \cap T) = (R \cdot S) \cap (R \cdot T) \quad \Leftarrow \quad (\mathit{ker}\, R) \cdot T \subseteq T \vee (\mathit{ker}\, R) \cdot S \subseteq S \tag{4.32}$$

since (for $R, T := \pi_1, \pi_1^\circ \cdot R$)

$$\begin{array}{ll} & \pi_1^\circ \cdot \pi_1 \cdot \pi_1^\circ \cdot R \subseteq \pi_1^\circ \cdot R \\ \Leftarrow & \qquad \{ \text{ monotonicity of composition} \} \\ & \pi_1^\circ \cdot \pi_1 \cdot \pi_1^\circ \subseteq \pi_1^\circ \\ \equiv & \qquad \{ \ \pi_1 \text{ is — as any function — difunctional } \} \\ & \text{TRUE} \end{array}$$

The alternative proof of (4.30) given below is based on (4.24) [29]:

$$
\begin{array}{cl}
 & R = \pi_1 \cdot \langle R, S \rangle \\
\equiv & \qquad \{ \text{ (4.24)} \} \\
 & R = R \cdot \delta\, S \\
\equiv & \qquad \{ \; R = R \cdot \delta\, R \text{ and composition of coreflexives is meet} \} \\
 & R = R \cdot (\delta\, R \cap \delta\, S) \\
\equiv & \qquad \{ \; R = R \cdot \delta\, R \} \\
 & \delta\, R \cap \delta\, S = \delta\, R \\
\equiv & \qquad \{ \; R \cap S = R \equiv R \subseteq S \} \\
 & \delta\, R \subseteq \delta\, S
\end{array}
$$

Our next goal is to calculate the concrete invariant induced by the nested join law (4.25).

The following fact will be of help in the calculation,

$$
usc(\overline{S} \cdot \Phi) \quad = \quad S \cdot (\Phi \times id) \tag{4.33}
$$

which can be inferred from (4.20) by *indirect equality* (see section A2, pag. 138):

$$
\begin{array}{cl}
 & usc(\overline{S} \cdot \Phi) \subseteq X \\
\equiv & \qquad \{ \text{ GC (4.20) } \} \\
 & \overline{S} \cdot \Phi \mathrel{\dot{\subseteq}} \overline{X} \\
\equiv & \qquad \{ \text{ shunting } (\overline{S} \text{ is a function) } \} \\
 & \Phi \subseteq \overline{S}^{\circ} \cdot \subseteq \cdot \overline{X} \\
\equiv & \qquad \{ \text{ going pointwise } \} \\
 & \langle \forall\, a \; : \; \Phi\, a : \; \overline{S}\, a \subseteq \overline{X}\, a \rangle \\
\equiv & \qquad \{ \text{ relational extensionality } \} \\
 & \langle \forall\, a \; : \; \Phi\, a : \; \langle \forall\, c, b \; :: \; c(\overline{S}\, a)b \Rightarrow c(\overline{X}\, a)b \rangle \rangle \\
\equiv & \qquad \{ \text{ logic and uncurrying, isomorphism (4.15)} \} \\
 & \langle \forall\, a, b \; : \; \Phi\, a : \; \langle \forall\, c \; :: \; c\, S(a,b) \Rightarrow c\, X(a,b) \rangle \rangle \\
\equiv & \qquad \{ \text{ back to pointfree } \} \\
 & \Phi \times id \subseteq S \setminus X \\
\equiv & \qquad \{ \text{ relational division } \} \\
 & S \cdot (\Phi \times id) \subseteq X \\
:: & \qquad \{ \text{ indirect equality} \} \\
 & usc(\overline{S} \cdot \Phi) = S \cdot (\Phi \times id)
\end{array}
$$

Also note the following fact:

$$\Phi \subseteq \Psi \times id \quad \equiv \quad \Psi \xleftarrow{\pi_1} \Phi \tag{4.34}$$

whose proof goes as follows:

$$
\begin{array}{ll}
 & \Phi \subseteq \Psi \times id \\
\equiv & \qquad \{ \text{ product} \} \\
 & \Phi \subseteq \langle \Psi \cdot \pi_1, \pi_2 \rangle \\
\equiv & \qquad \{ \text{ split; shunting; } \Phi \text{ is coreflexive } \} \\
 & \pi_1 \cdot \Phi \subseteq \Psi \cdot \pi_1 \ \wedge \ \pi_2 \cdot \Phi \subseteq \pi_2 \\
\equiv & \qquad \{ \text{ Reynolds (2.11)} \} \\
 & \Psi \xleftarrow{\pi_1} \Phi
\end{array}
$$

Now we are ready to calculate the invariant induced by (4.25):

$$
\begin{array}{ll}
 & \Delta_n(\bowtie_n (R, S)) = (R, S) \\
\equiv & \qquad \{ \text{ definition of } \Delta_n \text{ and of } \bowtie_n \} \\
 & (\pi_1 \cdot \langle R, \overline{S} \rangle, usc(\pi_2 \cdot \langle R, \overline{S} \rangle)) = (R, S) \\
\equiv & \qquad \{ \text{ (4.24) twice} \} \\
 & (R \cdot \delta\, \overline{S}, usc(\overline{S} \cdot \delta\, R)) = (R, S) \\
\equiv & \qquad \{ \text{ } \overline{S} \text{ is a function} \} \\
 & usc(\overline{S} \cdot \delta\, R) = S \\
\equiv & \qquad \{ \text{ equating simple relations (B.14) and } usc(\overline{S} \cdot \delta\, R) \subseteq S \text{ holds} \} \\
 & \delta\, S \subseteq \delta\, (usc(\overline{S} \cdot \delta\, R))
\end{array}
$$

We continue as follows,

$$
\begin{array}{cl}
& \delta\, S \subseteq \delta\,(usc(\overline{S} \cdot \delta\, R)) \\
\equiv & \quad \{\ (4.33)\} \\
& \delta\, S \subseteq \delta\,(S \cdot (\delta\, R \times id)) \\
\equiv & \quad \{\ \text{domain of composition}\} \\
& \delta\, S \subseteq \delta\,(\delta\, S \cdot (\delta\, R \times id)) \\
\equiv & \quad \{\ \text{coreflexives}\} \\
& \delta\, S \subseteq \delta\, S \cap (\delta\, R \times id) \\
\equiv & \quad \{\ \text{meet}\} \\
& \delta\, S \subseteq \delta\, R \times id \\
\equiv & \quad \{\ \ (4.34)\ \} \\
& \delta\, R \xleftarrow{\pi_1} \delta\, S
\end{array}
$$

Note how $\delta\, R \xleftarrow{\pi_1} \delta\, S$ expresses the concrete invariant in a quite "neat" way: π_1 has "type" $\delta\, R \longleftarrow \delta\, S$, that is, by projecting the domain of S by π_1 we obtain values in the domain of R. An alternative way to express this invariant is calculated below:

$$
\begin{array}{cl}
& \delta\, R \xleftarrow{\pi_1} \delta\, S \\
\equiv & \quad \{\ \text{Reynolds (2.11)}\ \} \\
& \pi_1 \cdot \delta\, S \subseteq \delta\, R \cdot \pi_1 \\
\equiv & \quad \{\ \text{shunting ; kernel}\} \\
& \delta\, S \subseteq \mathsf{ker}\,(\delta\, R \cdot \pi_1) \\
\equiv & \quad \{\ \text{introduce domain, since } \delta\, S \subseteq id\} \\
& \delta\, S \subseteq \delta\,(\delta\, R \cdot \pi_1) \\
\equiv & \quad \{\ \text{domain of composition}\ \} \\
& \delta\, S \subseteq \delta\,(R \cdot \pi_1) \\
\equiv & \quad \{\ \text{domain definition preorder: } X \preceq Y \equiv \delta\, X \subseteq \delta\, Y\ \} \\
& S \preceq R \cdot \pi_1
\end{array}
$$

The result obtained translates a simple pointwise argument [106] to relation algebra.

4.7 Laws concerning the Data Type of Relations

Recall that the data type of, simple relations, or partial functions from A to B is denoted by,

$$A \rightharpoonup B$$

The calculus SETS privileges $A \rightharpoonup B$ and its laws. However we can suspect that some of these laws can be generalized to arbitrary binary relations from A to B. We will denote by

$$A \rightsquigarrow B$$

the set of all binary relations from A to B. It's worth remembering the two elmentary isomorphisms (3.16) and

$$\begin{tikzcd} P(A \times B) \arrow[rr, bend left] \arrow[rr, phantom, "\cong"] & & 2^{A \times B} \arrow[ll, bend left] \end{tikzcd}$$

Since $A \rightharpoonup B \subseteq A \rightsquigarrow B$, we have

$$\begin{tikzcd} A \rightharpoonup B \arrow[rr, bend left, "mkr"] \arrow[rr, phantom, "\leq"] & & A \rightsquigarrow B \arrow[ll, bend left, "Mkf"] \end{tikzcd}$$

where

$$mkr(\sigma) \stackrel{\mathrm{def}}{=} \{(a, \sigma\ a) | a \in dom\ \sigma\}$$

and

$$Mkf(R) \stackrel{\mathrm{def}}{=} \{\pi_1(p) \mapsto \pi_2(p) | p \in R\}$$

subject to pre-condition

$$pre - Mkf(R) \stackrel{\mathrm{def}}{=} R \text{ is simple}$$

It's known that,

$$\begin{tikzcd} P(A) \arrow[rr, bend left, "set2fm"] \arrow[rr, phantom, "\cong"] & & A \rightharpoonup 1 \arrow[ll, bend left, "dom"] \end{tikzcd}$$

and

$$\begin{aligned} P(A) &\cong P(A \times 1) \\ &\cong A \rightsquigarrow 1 \end{aligned}$$

Thus

$$A \rightsquigarrow 1 \ = \ A \rightharpoonup 1$$

We are going to prove some laws concerning relations which extend known laws about partial functions.

Lemma 4.12 *The following isomorphism holds,*

$$\begin{tikzcd}(B+C) \rightsquigarrow A \arrow[rr, bend left, "unreither"] & \cong & (B \rightsquigarrow A) \times (C \rightsquigarrow A) \arrow[ll, bend left, "reither"]\end{tikzcd}$$

where

$$reither(R, S) \stackrel{\mathrm{def}}{=} [R, S]$$

is based on (2.25).

Alternatively, we may write

$$reither = \cup \cdot ((\cdot i_1^\circ) \times (\cdot i_2^\circ))$$

Its right inverse is,

$$unreither \stackrel{\mathrm{def}}{=} \langle (\cdot i_1), (\cdot i_2) \rangle$$

Proof: We are going to prove that these functions are each other inverses,

$$\begin{aligned}
& ([,] \cdot unreither)\ R \\
= & \qquad \{ \text{ application of } unreither \} \\
& [,](R \cdot i_1, R \cdot i_2) \\
= & \qquad \{ \text{ application of either} \} \\
& [R \cdot i_1, R \cdot i_2] \\
= & \qquad \{ (R\cdot) \text{ distributes over } \cup \} \\
& R \cdot [i_1, i_2] \\
= & \qquad \{ \text{ reflexion} \} \\
& R
\end{aligned}$$

and

$$\begin{aligned}
& unreither([,])(R, S) \\
= & \qquad \{ \text{ application of either} \} \\
& unreither([R, S]) \\
= & \qquad \{ \text{ application of } unreither \} \\
& ([R, S] \cdot i_1, [R, S] \cdot i_2) \\
= & \qquad \{ \text{ cancellation} \} \\
& (R, S)
\end{aligned}$$

$\square$

Lemma 4.13 *The following isomorphism generalizes law (4.13) to arbitrary binary relations:*

$$\begin{tikzcd} A \rightsquigarrow (B + C) \arrow[rr, bend left, "unrcojoin"] \arrow[rr, phantom, "\cong"] & & (A \rightsquigarrow B) \times (A \rightsquigarrow C) \arrow[ll, bend left, "rcojoin"] \end{tikzcd}$$

where:

$$rcojoin \stackrel{\mathrm{def}}{=} \cup \cdot ((i_1\cdot) \times (i_2\cdot))$$

$$unrcojoin \stackrel{\mathrm{def}}{=} \langle (i_1^\circ\cdot), (i_2^\circ\cdot) \rangle$$

Proof *This lemma results from its predecessor by applying converse which is an isomorphism on relations:*

$$\begin{tikzcd} A \rightsquigarrow B \arrow[rr, bend left, "()^\circ"] \arrow[rr, phantom, "\cong"] & & B \rightsquigarrow A \arrow[ll, bend left, "()^\circ"] \end{tikzcd}$$

Lemma 4.14 *The following isomorphism generalizes law (4.12) to arbitrary binary relations:*

$$\begin{tikzcd} (A \rightsquigarrow (B \times C) \arrow[rr, bend left, "unrjoin"] \arrow[rr, phantom, "\leq"] & & \left(\begin{array}{c} (A \rightsquigarrow B) \\ \times \\ (A \rightsquigarrow C) \end{array}, \subseteq \times \subseteq \right) \arrow[ll, bend left, "\bowtie"] \end{tikzcd}$$

The abstraction function is defined by,

$$\sigma \bowtie \tau \stackrel{\mathrm{def}}{=} \langle \sigma, \tau \rangle$$

In pointfree notation, there is,

$$\bowtie \stackrel{\mathrm{def}}{=} \cap \cdot ((\pi_1^\circ\cdot) \times (\pi_2^\circ\cdot))$$

The right inverse is defined by:

$$unrjoin \stackrel{\mathrm{def}}{=} \langle (\pi_1\cdot), (\pi_2\cdot) \rangle$$

Proof: This is Galois connection which generalizes Galois connection (4.19) defined for simple relations, to arbitrary relations. We shall prove that $unrjoin$ is the right inverse of $\bowtie$, generalizing proof in Proposition (4.2).

$$
\begin{array}{ll}
& \bowtie \cdot unrjoin \\
= & \quad \{ \textit{definitions} \} \\
& \cap \cdot ((\pi_1^\circ \cdot) \times (\pi_2^\circ \cdot)) \cdot \langle (\pi_1 \cdot), (\pi_2 \cdot) \rangle \\
= & \quad \{ \textit{absorption property} \} \\
& \cap \cdot \langle \pi_1^\circ \cdot \pi_1 \cdot, \pi_2^\circ \cdot \pi_2 \cdot \rangle \\
= & \quad \{ \textit{tabulations in } Rel \} \\
& id
\end{array}
$$

$\square$

4.8 Summary

This chapter developed a theory of data refinement around the relational calculus and (perfect) Galois connections. This includes calculation of concrete invariants induced by the Galois connected laws.

Some laws and respective proofs concern the data type of binary relations, which generalizes that of finite maps, i.e simple relations. The majority of them are isomorphism laws.

Isomorphisms involving simple relations, exponentials, finite sets and binary relations are introduced. The isormophisms form a calculus of their own with an independent compositionality result. The Maybe transpose, presented in section 3.1, is employed several times.

Chapter 5

Algorithmic Refinement

5.1 Introduction

This chapter addresses the process of formally converting abstract specifications of software operations into more concrete ones — ie. closer to machine level. The chapter focusses on the properties of the refinemnet ordering which underlies such processes and associated reasoning.

Suppose a component s of some piece of hardware gets faulty and needs to be replaced. Should no exact match be found off the shelf, the maintenance team will have to look around for *compatible* alternatives. What does *compatibility* mean in this context?

Let r be a candidate replacement for s and let the behaviour of both s and r be described by state-transition diagrams indicating, for each state a, the set of states reachable from a. So both s and r can be regarded as set-valued functions such that, for instance, component s may step from state a to state b iff $b \in (s\ a)$. Should $s\ a$ be empty, machine s will be in a deadlock and will fail.

The intuition behind r being a safe replacement for s — written $s \vdash r$ — is that, not only r should not fail where s does not,

$$\langle \forall\, a\ :\ (s\ a) \supset \emptyset :\ \emptyset \subset (r\ a) \rangle$$

but also that it should behave *"as s does"*. Wherever $(s\ a)$ is nonempty, there is some freedom for r to behave within such a set of choices: r is allowed be more deterministic than s. Altogether, one writes

$$s \vdash r \quad \stackrel{\mathrm{def}}{=} \quad \langle \forall\, a\ :\ (s\ a) \supset \emptyset :\ \emptyset \subset (r\ a) \subseteq (s\ a) \rangle \tag{5.1}$$

This definition of machine compatibility is nothing but a simplified version of that of *operation refinement* [135], the simplification being that one is not spelling out inputs and outputs and that, in general, the two machines s and r above need not share the same state space. This refinement ordering is standard in the discipline of *programming from specifications* [93] and can be found in various guises in the literature — see eg. references [135, 35, 142, 61] among many others. Reference [35] should be singled out for its detailed discussion of the lattice-theoretical properties of the $\vdash$ ordering.

Despite its wide adoption, this definition of $\vdash$ is not free from difficulties. It is a kind of ”meet of opposites”: non-determinism reduction suggests “smaller” behaviours while increase of definition suggests “larger” behaviours. This “anomaly” makes this standard notion of refinement less mathematically tractable than one would expect. For instance, Groves [61] points out that the principal operators in the Z schema calculus [142] are not monotonic with respect to $\vdash$-refinement. As a way of (partly) overcoming this problem, he puts forward an alternative characterisation of refinement based on the decomposition of $\vdash$ into two simpler relations,

$$s \vdash r \quad \equiv \quad \langle \exists\, t \; :: \; s \vdash_{pre} t \;\wedge\; t \vdash_{post} r \rangle \tag{5.2}$$

one per refinement concern: $\vdash_{pre}$ caters for increasing definition while $\vdash_{post}$ deals with decreasing non-determinism.

The same partition of the refinement relation was presented by the author of the current dissertation in [127] but the underlying theory was left unexplored. One of the aims of the current chapter is to extend and consolidate the work scattered in [127, 35, 61], focussing in particular in the last two, where some results are presented without proof and others are supported by either sketchy or convoluted arguments. The idea is to address the subject by reasoning in the pointfree relational calculus which is at the core of the *algebra of programming* [29, 12]. It should be noted that both [35, 61] already use some form of relational notation, somewhat mixed with the Z notation in the case of [61] or interpreted in terms of set-valued functions in [35]. The reasoning, however, is carried out at point-level, either involving predicate logic [61] or set-theory [35].

Our main contribution consists in resorting to the pointfree transform (*PF-transform* for short) all the way through, thus benefiting from not only its notation economy but also from the elegance of the associated reasoning style. It turns out that the theory becomes more general and simpler. Elegant expressions replace lengthy formulæ and easy-to-follow calculations replace pointwise proofs based on case analyses and natural language explanations.

Groves’ factorization (5.2) — which we will restate at pointfree level simply by writing

$$\vdash_{pre} \cdot \vdash_{post} \quad = \quad \vdash \quad = \quad \vdash_{post} \cdot \vdash_{pre} \tag{5.3}$$

— is central to the approach. Thanks to this factorization — which we calculate and justify in a way simpler than in [61] — we are able to justify facts which are stated but not proved in [35]. Among these, we present a new and detailed analysis, across the binary relation taxonomy, of the *lattice of specifications* proposed by [35].

As will be explained in chapter 6 and in the conclusions, this research is part of a broader initiative aiming at developing a PF-theory for coalgebraic refinement integrating earlier efforts already reported in [87, 21].

5.2 Warming up

According to the PF-transformation strategy announced in section 2.3, our first task will be to PF-transform (5.1). We first concentrate on transforming the test for non-deadlock states, which occurs twice in the formula, $(s\ a) \supset \emptyset$ and $(r\ a) \supset \emptyset$. A set is nonempty iff it contains at least one element. Therefore,

$$
\begin{array}{rcl}
(s\ a) \supset \emptyset & \equiv & \langle \exists\ x\ ::\ x \in (s\ a) \rangle \\
& \equiv & \quad \{\ \text{idempotence of } \wedge\ \} \\
& & \langle \exists\ x\ ::\ x \in (s\ a)\ \wedge\ x \in (s\ a) \rangle \\
& \equiv & \quad \{\ \text{(2.6) twice and converse}\ \} \\
& & \langle \exists\ x\ ::\ a(\in \cdot\, s)^{\circ}x\ \wedge\ x(\in \cdot\, s)a \rangle \\
& \equiv & \quad \{\ \text{introduce } b = a\text{ ; composition}\ \} \\
& & b = a\ \wedge\ b((\in \cdot\, s)^{\circ} \cdot (\in \cdot\, s))a \\
& \equiv & \quad \{\ \text{introduce kernel}\ \} \\
& & b = a\ \wedge\ b(\mathit{ker}\,(\in \cdot\, s))a
\end{array}
$$

Then we address the whole formula:

$$
\begin{array}{cl}
& s \vdash r \\
\equiv & \quad \{\ (5.1)\ \} \\
& \langle \forall\ a\ :\ (s\ a) \supset \emptyset :\ \emptyset \subset (r\ a) \subseteq (s\ a) \rangle \\
\equiv & \quad \{\ \text{expand } \emptyset \subset (r\ a) \subseteq (s\ a)\ \} \\
& \langle \forall\ a\ :\ (s\ a) \supset \emptyset :\ \emptyset \subset (r\ a)\ \wedge\ (r\ a) \subseteq (s\ a) \rangle \\
\equiv & \quad \{\ \text{expand tests for non-deadlock state and replace } (r\ a) \text{ by } (r\ b)\text{, cf. } b = a \} \\
& \langle \forall\ a, b\ :\ b = a\ \wedge\ b(\mathit{ker}\,(\in \cdot\, s))a :\ b = a\ \wedge\ b(\mathit{ker}\,(\in \cdot\, r))a\ \wedge\ (r\ b) \subseteq (s\ a) \rangle \\
\equiv & \quad \{\ \delta\, R = \mathit{ker}\, R \cap id\text{ , for every } R\ \} \\
& \langle \forall\ a, b\ :\ b(\delta\,(\in \cdot\, s))a :\ b(\delta\,(\in \cdot\, r))a\ \wedge\ (r\ b) \subseteq (s\ a) \rangle \\
\equiv & \quad \{\ \text{expand set-theoretic inclusion}\ \} \\
& \langle \forall\ a, b\ :\ b(\delta\,(\in \cdot\, s))a :\ b(\delta\,(\in \cdot\, r))a\ \wedge\ \langle \forall\ c\ :\ c \in (r\ b) :\ c \in (s\ b) \rangle \rangle \\
\equiv & \quad \{\ \text{(2.6) twice ; then introduce left-division (2.18)}\ \} \\
& \langle \forall\ a, b\ :\ b(\delta\,(\in \cdot\, s))a :\ b(\delta\,(\in \cdot\, r))a\ \wedge\ b((\in \cdot\, r) \setminus (\in \cdot\, s))a \rangle \\
\equiv & \quad \{\ \text{remove points ; relational inclusion and meet}\ \} \\
& \delta\,(\in \cdot\, s) \subseteq \delta\,(\in \cdot\, r) \cap ((\in \cdot\, r) \setminus (\in \cdot\, s)) \\
\equiv & \quad \{\ \text{remove membership by defining } R = \in \cdot\, r \text{ and } S = \in \cdot\, S \} \\
& \delta\, S \subseteq \delta\, R \cap (R \setminus S)
\end{array}
$$

Function s (resp. r) can be identified with the *power-transpose* (3.1) of binary relation S (resp. R). Since transposition is an isomorphism, we can safely lift our original ordering on set-valued state-transition functions to state-transition relations and establish the relational PF-transform of (5.1) as follows:

$$S \vdash R \quad \equiv \quad \delta\, S \subseteq (R \setminus S) \cap \delta\, R \tag{5.4}$$

which converts to

$$S \vdash R \quad \equiv \quad (\delta\, S \subseteq \delta\, R) \;\wedge\; (R \cdot \delta\, S \subseteq S) \tag{5.5}$$

once universal property of meet and Galois connection of left-division (2.15) are taken into account.

Most definitions of the refinement ordering in the literature — eg. [135, 35, 142, 61] — are pointwise variants of (5.5). The calculations above show these to be equivalent to our starting version (5.1), which instantiates a "coalgebraic pattern" favoured in automata theory and coalgebraic refinement, as will be seen in chapter 6.

It is easy to see that the source and target types of both S, R in (5.5) need not be the same:

$$\begin{tikzcd} A \arrow[dr, "R"'] & & A \arrow[ll, "\delta\, S"'] \arrow[dl, "S"] \\ & B & \end{tikzcd}$$

So, PF-transformed $S \vdash R$ covers other refinement situations, namely that of an *implicit specification* [75] S being refined by some function f,

$$S \vdash f \quad \equiv \quad \delta\, S \subseteq f^{\circ} \cdot S$$

whereby — back to points and thanks to (2.6) — we obtain, in classical "VDM-speak"

$$\forall a.\text{pre-}S(a) \Rightarrow \text{post-}S(f\ a, a) \tag{5.6}$$

which is nothing but the *implicit function specification* proof-rule given by [75].

It is in this (wider) context that the $\vdash$ ordering is presented in [35], where it is called the *less-defined* relation on specifications and is shown to be a semi-lattice universally lower-bounded by the empty specification $\perp$. The proof that it is a partial order is telegram-like in the paper. By contrast, the existence of a greatest lower bound (*glb*) is the subject of a proposition proved in typical *invent & verify* style — a *glb* definition is guessed first, which is then shown to be a lower bound and finally proved to be maximal among all lower bounds.

To illustrate the shift from *verification* to *calculation* brought forth by the PF-transform, we will *calculate* the *glb* of $\vdash$ (denoted $\sqcap$) as the (unique) solution to universal property

$$X \vdash R \sqcap S \quad \equiv \quad X \vdash R \;\wedge\; X \vdash S \tag{5.7}$$

Let us solve this equation for unknown $\sqcap$:

$$
\begin{array}{ll}
 & X \vdash R \sqcap S \\
\equiv & \qquad \{ \text{ (5.7) } \} \\
 & X \vdash R \;\wedge\; X \vdash S \\
\equiv & \qquad \{ \text{ (5.4) twice; composition of coreflexives is intersection } \} \\
 & \delta\, X \subseteq ((R \setminus X \cap S \setminus X)) \cap \delta\, R \cdot \delta\, S \\
\equiv & \qquad \{ \text{ (2.19)} \} \\
 & \delta\, X \subseteq (R \cup S) \setminus X \cap \delta\, R \cdot \delta\, S \\
\equiv & \qquad \{ \text{ (2.20) for } R, S, \Phi := R \cup S, X, \delta\, R \cdot \delta\, S \ \} \\
 & \delta\, X \subseteq (((R \cup S) \cdot \delta\, R \cdot \delta\, S) \setminus X) \cap (\delta\, R \cdot \delta\, S) \\
\equiv & \qquad \{ \ \delta\, ((R \cup S) \cdot \delta\, R \cdot \delta\, S) = \delta\, R \cdot \delta\, S \text{ (coreflexives) ; (5.4) } \} \\
 & X \vdash ((R \cup S) \cdot \delta\, R \cdot \delta\, S) \\
:: & \qquad \{ \text{ indirect equality on partial order } \vdash \ \} \\
 & R \sqcap S = (R \cup S) \cdot \delta\, R \cdot \delta\, S
\end{array}
$$

Thus we have deduced

$$R \sqcap S \quad = \quad (R \cup S) \cdot \delta\, R \cdot \delta\, S \tag{5.8}$$

which, back to points (5.1), will look like

$$(r \sqcap s)a \quad = \quad if\ (r\ a) = \emptyset \vee (s\ a) = \emptyset\ then\ \emptyset\ else\ (r\ a) \cup (s\ a) \tag{5.9}$$

where r (resp. s) is the power-transpose of R (resp. S). (The reader is invited to calculate (5.9) as solution to (5.7) by directly resorting to the pointwise definition of $\vdash$ (5.1) instead of (5.4).)

5.3 Refinement sub-relations

Recall the two conjuncts of (5.5), $\delta\, S \;\subseteq\; \delta\, R$ and $R \cdot \delta\, S \;\subseteq\; S$. Groves [61] freezes the former in defining a sub-relation $\vdash_{post}$ of $\vdash$,

$$S \vdash_{post} R \quad \equiv \quad S \vdash R \;\wedge\; \delta\, R \subseteq \delta\, S \tag{5.10}$$

where extra clause $\delta\, R \subseteq \delta\, S$ prevents definition increase (by antisymmetry). Similarly, he puts forward another sub-relation $\vdash_{pre}$ of $\vdash$,

$$S \vdash_{pre} R \equiv S \vdash R \;\wedge\; S \subseteq R \cdot \delta\, S \tag{5.11}$$

where extra clause $S \subseteq R \cdot \delta\, S$ prevents from increasing determinacy.

How useful are these sub-orderings? Most of this chapter will be devoted to exploiting the underlying theory and showing them to be useful beyond their original context of definition [61]. First of all, facts

$$\vdash_{pre} \;\subseteq\; \vdash \tag{5.12}$$
$$\vdash_{post} \;\subseteq\; \vdash \tag{5.13}$$

are immediate consequences of definitions (5.10,5.11) above.

That both $\vdash_{pre}$ and $\vdash_{post}$ can be expressed independently of $\vdash$ is simple to calculate, first for $\vdash_{pre}$,

$$\begin{array}{cl} & S \vdash_{pre} R \\ \equiv & \qquad \{ \text{ (5.11) and (5.5) ; antisymmetry } \} \\ & R \cdot \delta\, S = S \;\wedge\; \delta\, S \subseteq \delta\, R \\ \equiv & \qquad \{ \text{ switch to conditions (2.17) } \} \\ & R \cdot \delta\, S = S \;\wedge\; ! \cdot S \subseteq ! \cdot R \\ \equiv & \qquad \{ \text{ substitution of } S \text{ by } R \cdot \delta\, S \} \\ & R \cdot \delta\, S = S \;\wedge\; ! \cdot R \cdot \delta\, S \subseteq ! \cdot R \\ \equiv & \qquad \{ \;\delta\, S \text{ is coreflexive } (\delta\, S \subseteq id) \text{ ; monotonicity of composition } \} \\ & R \cdot \delta\, S = S \;\wedge\; \textsc{True} \\ \equiv & \qquad \{ \text{ trivia } \} \\ & R \cdot \delta\, S = S \end{array}$$

and then for $\vdash_{post}$:

$$\begin{array}{cl} & S \vdash_{post} R \\ \equiv & \qquad \{ \text{ (5.10) and (5.5)} \} \\ & R \cdot \delta\, S \subseteq S \;\wedge\; \delta\, R = \delta\, S \\ \equiv & \qquad \{ \text{ substitution of } \delta\, S \text{ by } \delta\, R \} \\ & R \cdot \delta\, R \subseteq S \;\wedge\; \delta\, R = \delta\, S \\ \equiv & \qquad \{ \; R \cdot \delta\, R = R \} \\ & R \subseteq S \;\wedge\; \delta\, R = \delta\, S \end{array}$$

Let us record these results, which are the PF-counterparts to laws 4.3 and 4.4 in [61], respectively,

$$S \vdash_{pre} R \;\equiv\; R \cdot \delta\, S = S \tag{5.14}$$
$$S \vdash_{post} R \;\equiv\; R \subseteq S \;\wedge\; \delta\, R = \delta\, S \tag{5.15}$$

noting that (5.15) can be compressed into the terser notation which follows:

$$\vdash_{post} \;=\; \subseteq^{\circ} \cap\, \mathit{ker}\, \delta \tag{5.16}$$

What does it mean to impose $\vdash_{pre}$ and $\vdash_{post}$ at the same time? We calculate:

$$
\begin{array}{cl}
& S \vdash_{pre} R \;\wedge\; S \vdash_{post} R \\
\equiv & \quad \{\ (5.14), (5.15)\ \} \\
& R \cdot \delta\, S = S \;\wedge\; \delta\, S = \delta\, R \;\wedge\; R \subseteq S \\
\equiv & \quad \{\ \text{substitution of } \delta\, S \text{ by } \delta\, R\ \} \\
& R \cdot \delta\, R = S \;\wedge\; \delta\, R = \delta\, R \;\wedge\; R \subseteq S \\
\equiv & \quad \{\ \text{property } R = R \cdot \delta\, R \} \\
& R = S \;\wedge\; R \subseteq S \\
\equiv & \quad \{\ R = S \Rightarrow R \subseteq S\ \} \\
& R = S
\end{array}
$$

This result, which is law 4.7 in [61], can be expressed in less symbols by writing

$$\vdash_{pre} \cap \vdash_{post} \quad = \quad id \tag{5.17}$$

whose "antisymmetric pattern" captures the opposition between the components $\vdash_{pre}$ and $\vdash_{post}$ of $\vdash$: to increase determinism only *and* definition only at the same time is contradictory. This relative antisymmetry between $\vdash_{pre}$ and $\vdash_{post}$ can also be inferred from facts

$$S \vdash_{post} R \quad \Rightarrow \quad R \subseteq S \tag{5.18}$$

$$S \vdash_{pre} R \quad \Rightarrow \quad S \subseteq R \tag{5.19}$$

the former arising immediately from (5.15) and the latter holding by transitivity: $S \vdash_{pre} R$ implies $S \subseteq R \cdot \delta\, S$ and $R \cdot \delta\, S \subseteq R$ holds.

Groves [61] glosses over the proofs that $\vdash_{pre}$ and $\vdash_{post}$ are partial orders. These are immediate at PF-transform level: that $\vdash_{post}$ is a partial order is obvious from (5.16), since the meet of a partial order ($\subseteq$) with an equivalence (*ker* δ) is a partial order. In fact, a partial order (say R) and an equivalence (say S) only disagree wrt. symmetry. It turns up that $R \cap S$ is anti-symmetric:

$$
\begin{array}{cl}
& (R \cap S) \cap (R \cap S)^\circ \subseteq id \\
\equiv & \quad \{\ \cap \text{ commutes with converse}\ \} \\
& R \cap R^\circ \cap S \cap S^\circ \subseteq id \\
\Leftarrow & \quad \{\ \text{substitution}\ ;\ S \cap S = S\ \} \\
& R \cap R^\circ \cap S \subseteq id \;\wedge\; S = S^\circ \\
\Leftarrow & \quad \{\ R \cap S \subseteq S \} \\
& R \cap R^\circ \subseteq id \;\wedge\; S = S^\circ \\
\equiv & \quad \{\ R \text{ assumed anti-symmetric and } S \text{ symmetric}\ \} \\
& \text{TRUE}
\end{array}
$$

Next we show that $\vdash_{pre}$ is a partial order. Substitution $R := S$ in (5.14) leads to reflexivity at once. Antisymmetry stems from (5.19), since $\subseteq$ is antisymmetric and smaller than antisymmetric is antisymmetric:

Lemma 5.1 *Smaller than antisymmetric is antisymmetric.*
Proof:

$$
\begin{array}{ll}
 & R \subseteq S \ \wedge\ S \cap S^\circ \subseteq id \\
\equiv & \quad \{ \textit{ monotonicity of converse} \} \\
 & R \subseteq S \ \wedge\ R^\circ \subseteq S^\circ \ \wedge\ S \cap S^\circ \subseteq id \\
\Rightarrow & \quad \{ \textit{ monotonicity of meet} \} \\
 & R \cap R^\circ \subseteq S \cap S^\circ \ \wedge\ S \cap S^\circ \subseteq id \\
\Rightarrow & \quad \{ \textit{ transitivity } \} \\
 & R \cap R^\circ \subseteq id
\end{array}
$$

□

In this way, only transitivity

$$S \vdash_{pre} R \ \wedge\ R \vdash_{pre} T \Rightarrow S \vdash_{pre} T$$

requires an explicit proof:

$$
\begin{array}{ll}
 & S \vdash_{pre} R \ \wedge\ R \vdash_{pre} T \\
\equiv & \quad \{ \ (5.14) \ \} \\
 & R \cdot \delta\, S = S \ \wedge\ T \cdot \delta\, R = R \\
\Rightarrow & \quad \{ \text{ substitution } R := T \cdot \delta\, R \ \} \\
 & (T \cdot \delta\, R) \cdot \delta\, S = S \\
\equiv & \quad \{ \text{ composition of coreflexives is meet} \} \\
 & T \cdot (\delta\, R \cap \delta\, S) = S \\
\equiv & \quad \{ \ S \subseteq R \text{ (5.19) and thus } \delta\, S \subseteq \delta\, R \ \} \\
 & T \cdot \delta\, S = S \\
\equiv & \quad \{ \ (5.14) \ \} \\
 & S \vdash_{pre} T
\end{array}
$$

5.4 Factorization of the refinement relation

We proceed to showing that the sequential composition of subrelations $\vdash_{pre}$ and $\vdash_{post}$ is — in any order — the refinement relation $\vdash$ itself. As we shall see, this is where our calculational style differs more substantially from that of [61].

That $\vdash_{pre}$ and $\vdash_{post}$ are *factors* of $\vdash$ — that is,

$$\vdash_{post} \cdot \vdash_{pre} \quad \subseteq \quad \vdash \tag{5.20}$$
$$\vdash_{pre} \cdot \vdash_{post} \quad \subseteq \quad \vdash \tag{5.21}$$

— is obvious, recall (5.12, 5.13) and composition monotonicity. So we will focus on the converse facts

$$\vdash \quad \subseteq \quad \vdash_{pre} \cdot \vdash_{post} \tag{5.22}$$
$$\vdash \quad \subseteq \quad \vdash_{post} \cdot \vdash_{pre} \tag{5.23}$$

As earlier on, instead of postulating these decompositions and then proving them, we will calculate (deduce) them. Two auxiliary results will be required:

$$S \vdash_{post} S \cap R \quad \equiv \quad \delta\, S = \delta\,(R \cap S) \tag{5.24}$$
$$S \vdash_{pre} S \cup R \quad \equiv \quad R \cdot \delta\, S \subseteq S \tag{5.25}$$

The proof of (5.24) is immediate from the definition of $\vdash_{post}$ (5.15). That of (5.25) follows:

$$\begin{array}{cl}
& S \vdash_{pre} S \cup R \\
\equiv & \qquad \{ \text{ definition of } \vdash_{pre} \} \\
& (S \cup R) \cdot \delta\, S = S \\
\equiv & \qquad \{ \text{ } (\cdot \delta\, S) \text{ is a lower adjoint (2.15) } \} \\
& (S \cdot \delta\, S) \cup (R \cdot \delta\, S) = S \\
\equiv & \qquad \{ \; S \cdot \delta\, S = S \} \\
& S \cup R \cdot \delta\, S = S \\
\equiv & \qquad \{ \; A \cup B = B \equiv A \subseteq B \} \\
& R \cdot \delta\, S \subseteq S
\end{array}$$

We are now ready to calculate (5.22):

$$
\begin{array}{cl}
 & S \vdash R \\
\equiv & \qquad \{ \text{ (5.5) } \} \\
 & R \cdot \delta\, S \subseteq S \ \wedge\ \delta\, S \subseteq \delta\, R \\
\equiv & \qquad \{ A \cup B = B \equiv A \subseteq B \} \\
 & R \cdot \delta\, S \subseteq S \ \wedge\ (\delta\, S) \cup (\delta\, R) = \delta\, R \\
\equiv & \qquad \{ \delta \text{ is a lower adjoint } \} \\
 & R \cdot \delta\, S \subseteq S \ \wedge\ \delta\, (S \cup R) = \delta\, R \\
\equiv & \qquad \{ \text{ (5.24), since } R = R \cap (S \cup R) \ \} \\
 & R \cdot \delta\, S \subseteq S \ \wedge\ (S \cup R) \vdash_{post} R \cap (S \cup R) \\
\equiv & \qquad \{ \text{ (5.25) and } R = R \cap (S \cup R) \ \} \\
 & (S \vdash_{pre} S \cup R) \ \wedge\ (S \cup R) \vdash_{post} R \\
\Rightarrow & \qquad \{ \text{ logic } \} \\
 & \langle \exists\, T \ :: \ S \vdash_{pre} T \ \wedge\ T \vdash_{post} R \rangle \\
\equiv & \qquad \{ \text{ composition } \} \\
 & S (\vdash_{pre} \cdot \vdash_{post}) R
\end{array}
$$

Concerning (5.23):

$$
\begin{array}{cl}
 & S \vdash R \\
\Rightarrow & \qquad \{ \text{ since } \ S \vdash R \ \Rightarrow \ \delta\, S = \delta\, (S \cap R) \ \text{ (B.19); (5.5) } \} \\
 & \delta\, S = \delta\, (S \cap R) \ \wedge\ R \cdot \delta\, S \subseteq S \\
\equiv & \qquad \{ \ \cap\text{-universal and } \delta\, S \text{ is coreflexive } \} \\
 & \delta\, S = \delta\, (S \cap R) \ \wedge\ R \cdot \delta\, S \subseteq S \cap R \\
\equiv & \qquad \{ \text{ substitution } \} \\
 & \delta\, S = \delta\, (S \cap R) \ \wedge\ R \cdot \delta\, (S \cap R) \subseteq S \cap R \\
\equiv & \qquad \{ \text{ (5.25) } \} \\
 & \delta\, S = \delta\, (S \cap R) \ \wedge\ (S \cap R) \vdash_{pre} (S \cap R) \cup R \\
\equiv & \qquad \{ \text{ (5.24) and } S \cap R \subseteq R \ \} \\
 & (S \vdash_{post} S \cap R) \ \wedge\ (S \cap R) \vdash_{pre} R \\
\Rightarrow & \qquad \{ \text{ logic } \} \\
 & \langle \exists\, T \ :: \ S \vdash_{post} T \ \wedge\ T \vdash_{pre} R \rangle \\
\equiv & \qquad \{ \text{ composition } \} \\
 & S (\vdash_{post} \cdot \vdash_{pre}) R
\end{array}
$$

In summary, we have the two alternative ways to factor the refinement relation announced in (5.3). This embodies laws 4.8 and 4.9 of [61], where they are proved in first-order logic requiring negation and consistency [1]. These requirements, which have no counterpart in our calculations above, should be regarded as spurious.

5.5 Taking advantage of the factorization

Factorizations such as that given by (5.3) are very useful in mathematics in general. For our purposes, the rôle of (5.3) is three-fold. On the one hand, properties of the composition — eg. transitivity, reflexivity — can be easily inferred from similar properties of factors $\vdash_{pre}$ and $\vdash_{post}$. On the other hand, one can look for results with hold for the individual factors $\vdash_{pre}$ and/or $\vdash_{post}$ and do not hold (in general) for $\vdash$. For instance, meet ($R \cap S$) is $\vdash_{pre}$-monotonic but not $\vdash$-monotonic (law 5.1 in [61]). This aspect of the factorization is of practical value and in fact the main motivation in [61]: complex refinement steps can be factored in *less big a gap* ones involving only one factor $\vdash_{pre}$ (resp. $\vdash_{post}$) and $\vdash_{pre}$ (resp. $\vdash_{post}$) monotonic operators.

The following lemma presents our calculation of the two main monotonicity facts of [61]:

$$S \vdash_{pre} R \;\wedge\; T \vdash_{pre} U \quad\Rightarrow\quad S \cap T \vdash_{pre} R \cap U \tag{5.26}$$

$$S \vdash_{post} R \;\wedge\; T \vdash_{post} U \quad\Rightarrow\quad S \cup T \vdash_{post} R \cup U \tag{5.27}$$

(These are laws 5.1 and 5.4 in [61].) Again we stress on the fact that our pointfree calculations don't require negation.

Lemma 5.2 ($\vdash_{pre}$/$\vdash_{post}$-monotonicity of meet and join) *Meet is $\vdash_{pre}$- monotonic (5.26), and join is $\vdash_{post}$- monotonic (5.27).*

Proof: Concerning (5.26), we investigate the $\vdash_{pre}$-monotonicity of section $(\cap T)$ (this is enough, since meet is commutative):

[1] In [61, 35], two relations R and S are regarded as *consistent* iff $\delta\,(R \cap S) = (\delta\, R) \cap (\delta\, S)$ holds.

$$
\begin{array}{ll}
& S \cap T \vdash_{pre} R \cap T \\
\equiv & \quad \{ \text{ (5.14)} \} \\
& (R \cap T) \cdot \delta\, (S \cap T) = S \cap T \\
\equiv & \quad \{ \ \delta\, (S \cap T) \subseteq \delta\, S\text{; meet of coreflexives is composition } \} \\
& (R \cap T) \cdot (\delta\, S) \cdot \delta\, (S \cap T) = S \cap T \\
\equiv & \quad \{ \text{ (B.18), since } \delta\, S \text{ is coreflexive } \} \\
& ((R \cdot \delta\, S) \cap T) \cdot \delta\, (S \cap T) = S \cap T \\
\Leftarrow & \quad \{ \text{ substitution } \} \\
& S = R \cdot \delta\, S \ \wedge \ (S \cap T) \cdot \delta\, (S \cap T) = S \cap T \\
\equiv & \quad \{ \text{ (5.14) and } R \cdot \delta\, R = R \ \} \\
& S \vdash_{pre} R
\end{array}
$$

Concerning (5.27), we proceed in the same way by proving

$$S \vdash_{post} R \quad \Rightarrow \quad S \cup T \vdash_{post} R \cup T$$

that is,

$$(\cup T) \cdot \vdash_{post} \quad \subseteq \quad \vdash_{post} \cdot (\cup T)$$

at pointfree level:

$$
\begin{array}{ll}
& (\cup T) \cdot \vdash_{post} \\
= & \quad \{ \text{ definition 5.16 } \} \\
& (\cup T) \cdot (\subseteq^\circ \cap (\mathit{ker}\, \delta)) \\
\subseteq & \quad \{ \ ((\cup T) \cdot) \text{ is monotonic } \} \\
& ((\cup T) \cdot \subseteq^\circ) \cap ((\cup T) \cdot \mathit{ker}\, \delta) \\
\subseteq & \quad \{ \ (\cup T) \text{ is monotonic and } \mathit{ker}\, \delta \subseteq \mathit{ker}\, (\delta \cdot (\cup T)) \ \} \\
& (\subseteq^\circ \cdot (\cup T)) \cap ((\cup T) \cdot \mathit{ker}\, (\delta \cdot (\cup T))) \\
\subseteq & \quad \{ \text{ (B.20) } \} \\
& (\subseteq^\circ \cdot (\cup T)) \cap (\mathit{ker}\, (\delta) \cdot (\cup T)) \\
= & \quad \{ \text{ (B.15) } \} \\
& (\subseteq^\circ \cap \mathit{ker}\, \delta) \cdot (\cup T) \\
= & \quad \{ \text{ definition 5.16 } \} \\
& \vdash_{post} \cdot (\cup T)
\end{array}
$$

□

We know that composition is not $\vdash$-monotonic, cf. [51]. But a section of composition is monotonic with respect to $\vdash_{post}$,

$$S \vdash_{post} R \quad \Rightarrow \quad S \cdot T \vdash_{post} R \cdot T \tag{5.28}$$

The proof of this result follows,

$$\begin{array}{ll} & S \vdash_{post} R \\ \equiv & \qquad \{ \text{ by definition} \} \\ & R \subseteq S \wedge \delta\, S = \delta\, R \\ \Rightarrow & \qquad \{ \text{ monotonicity of composition and equality} \} \\ & R \cdot T \subseteq S \cdot T \wedge \delta\, (\delta\, S) \cdot T = \delta\, (\delta\, R) \cdot T \\ \equiv & \qquad \{ \text{ property of } \delta \} \\ & R \cdot T \subseteq S \cdot T \wedge \delta\, S \cdot T = \delta\, R \cdot T \\ \equiv & \qquad \{ \text{ by definition} \} \\ & S \cdot T \vdash_{post} R \cdot T \end{array}$$

Composition is not full monotonic with respect to $\vdash_{post}$ since it's not commutative.

In the sequel, we study under which conditions monotonicity of composition holds with respect to $\vdash$

Lemma 5.3 *If T is entire then*

$$S \vdash R \Rightarrow T \cdot S \vdash T \cdot R$$

holds:

$$\begin{array}{ll} & T \cdot S \vdash T \cdot R \\ \equiv & \qquad \{ \textit{ by definition of } \vdash \} \\ & (T \cdot S \cdot \delta\, (T \cdot R) \subseteq T \cdot R) \;\wedge\; \delta\, (T \cdot S) \subseteq \delta\, (T \cdot R) \\ \equiv & \qquad \{ \; T \textit{ is entire} \} \\ & (T \cdot S \cdot \delta\, R \subseteq T \cdot R) \wedge \delta\, S \subseteq \delta\, R \\ \Leftarrow & \qquad \{ \; (T \cdot) \textit{ is an adjoint, therefore monotonic} \} \\ & S \cdot \delta\, R \subseteq R \wedge \delta\, S \subseteq \delta\, R \\ \equiv & \qquad \{ \textit{ definition} \} \\ & S \vdash R \end{array}$$

Lemma 5.4 (Functions)

$$S \vdash f \Rightarrow S \cdot T \vdash f \cdot T$$

Proof:

$$
\begin{array}{cl}
 & S \cdot T \vdash f \cdot T \\
\equiv & \quad \{ \textit{ by definition of} \vdash \} \\
 & (S \cdot T \cdot \delta\,(f \cdot T) \subseteq f \cdot T) \ \wedge \ \delta\,(S \cdot T) \subseteq \delta\,(f \cdot T) \\
\equiv & \quad \{ \textit{functions} \} \\
 & (S \cdot T \cdot \delta\, T \subseteq f \cdot T) \wedge \delta\, S \cdot T \subseteq \delta\, T \\
\equiv & \quad \{ \ R = R \cdot \delta\, R \} \\
 & (S \cdot T \subseteq f \cdot T) \wedge \delta\,(S \cdot T) \subseteq \delta\, T \\
\equiv & \quad \{ \ \delta\,(R \cdot S) \subseteq \delta\, S \} \\
 & S \cdot T \subseteq f \cdot T \\
\Leftarrow & \quad \{ \ (\cdot T) \textit{ is an adjoint, therefore monotonic} \} \\
 & S \cdot \delta\, f \subseteq f \\
\equiv & \quad \{ \textit{ definitions} \} \\
 & S \vdash f
\end{array}
$$

A third kind of advantage of factorization (5.3) has been left uneunexploited in [61]: the fact that it makes it easy to analyse the (semi-)lattice of operations ordered by $\vdash$ [35], in particular concerning the behaviour of factors $\vdash_{pre}$ and $\vdash_{post}$ for some of the relation subclasses depicted in the diagram of Fig. 2.1. For instance, if by construction one knows that the operation under refinement is *simple* (vulg. a partial function), one can safely replace $\vdash$ by the appropriate factors tabulated in

Binary relation sub-class	$\vdash_{post}$	$\vdash_{pre}$	$\vdash$	
Entire relations	$\subseteq^\circ$	id	$\subseteq^\circ$	(a)
Simple relations	id	$\subseteq$	$\subseteq$	(b)
Functions	id	id	id	(c)

(5.29)

Let us justify (5.29): $\vdash_{pre} = id$ in case (5.29a) follows directly from (5.14), $\vdash_{pre}$-refinement of entire relations is the identity relation on (entire) relations,

$$S \vdash_{pre} R \quad \equiv \quad R = S$$

in which case equation (5.3) yields $\vdash \, = \, \vdash_{post}$. Moreover, $\vdash_{post} \; = \; \subseteq^\circ$ holds since domain (δ) is a constant function within the class of *entire* relations and thus *ker* $\delta = \top$ in (5.16). The proof of (5.29b) in the case of $\vdash_{post} = id$, follows from the fact that (5.15) restricted to simple relations establishes equality at once:

$$
\begin{array}{ll}
& S \vdash_{post} R \\
\equiv & \quad \{ \text{ (5.15) ; anti-symmetry } \} \\
& R \subseteq S \ \wedge\ \delta\, R \subseteq \delta\, S \ \wedge\ \delta\, S \subseteq \delta\, R \\
\equiv & \quad \{ \text{ (B.14) } \} \\
& R = S \ \wedge\ \delta\, R \subseteq \delta\, S \\
\equiv & \quad \{ \text{ second conjunct implied by first } \} \\
& R = S
\end{array}
$$

Concerning $\vdash_{pre} =\subseteq$, our calculation to follow will rely on relaxing function f to a simple relation S in the *shunting* rules in (2.15), leading to rules [99]

$$
S \cdot R \subseteq T \quad \equiv \quad (\delta\, S) \cdot R \ \subseteq\ S^{\circ} \cdot T \tag{5.30}
$$

$$
R \cdot S^{\circ} \subseteq T \quad \equiv \quad R \cdot \delta\, S \ \subseteq\ T \cdot S \tag{5.31}
$$

which, however, are not Galois connections. We reason:

$$
\begin{array}{ll}
& S \vdash_{pre} R \\
\equiv & \quad \{ \text{ (5.14) ; anti-symmetry} \} \\
& R \cdot \delta\, S \subseteq S \ \wedge\ S \subseteq R \cdot \delta\, S \\
\equiv & \quad \{ \text{ shunt on simple } R \text{ (5.30) and } S \text{ (5.31) ; } S = S \cdot \delta\, S \ \} \\
& \delta\, R \cdot S^{\circ} \subseteq R^{\circ} \ \wedge\ S \cdot \delta\, S \subseteq R \cdot \delta\, S \\
\equiv & \quad \{ \text{ converses } \} \\
& S \cdot \delta\, R \subseteq R \ \wedge\ S \cdot \delta\, S \subseteq R \cdot \delta\, S \\
\equiv & \quad \{ \ \delta\, S \subseteq \delta\, R, \text{ cf. (5.19) and monotonicity of } \delta \ \} \\
& S \subseteq R \ \wedge\ S \cdot \delta\, S \subseteq R \cdot \delta\, S \\
\equiv & \quad \{ \text{ first conjunct implies the second (monotonicity) } \} \\
& S \subseteq R
\end{array}
$$

Finally, (5.29(c)) follows from functions being entire and simple at the same time (Fig. 2.1).

Union of simple S and R is simple if both are simple and $S \cdot (\delta\, R) \subseteq R$, which is equivalent to $R \cdot (\delta\, S) \subseteq S$ thanks to (5.30 , 5.31). This is easy to calculate:

$$
\begin{array}{ll}
& S \cup R \text{ is simple} \\
\equiv & \quad \{ \text{simple relation definition, see table (2.1)} \} \\
& (S \cup R) \cdot (S \cup R)^\circ \subseteq id \\
\equiv & \quad \{ \cdot R \text{ and } (R \cdot) \text{ are lower adjoints and universal property of join} \} \\
& S \cdot S^\circ \subseteq id \ \wedge \ S \cdot R^\circ \subseteq id \ \wedge \ R \cdot S^\circ \subseteq id \ \wedge \ R \cdot R^\circ \subseteq id \\
\equiv & \quad \{ \text{ converses } \} \\
& S \text{ and } R \text{ are simple } \wedge \ S \cdot R^\circ \subseteq id \\
\equiv & \quad \{ \text{ (5.31) } \} \\
& S \text{ and } R \text{ are simple } \wedge \ S \cdot (\delta\, R) \subseteq R
\end{array}
$$

So $R \overset{pre}{\sqcap} S$ is not ensured for simple relations. A comment on the *glb* of $\vdash_{pre}$ restricted to simple relations, ie. deterministic but possibly failing operations (partial functions): pointfree calculation yields $R \sqcap S = R \cap S$ in this case, which agrees with $\subseteq$ in (5.29b) but contrasts to factor $R \cup S$ in (5.8). It can be easily calculated that simplicity of $R \sqcap S$ (5.8) is equivalent to both R, S being simple and $R \cdot S^\circ \subseteq id$, which is equivalent, thanks to (5.31), to $R \cdot \delta\, S \subseteq S$, itself equivalent to $S \cdot \delta\, R \subseteq R$:

$$
\begin{array}{ll}
& R \cdot \delta\, S \subseteq S \\
\equiv & \quad \{ \text{ law (5.31) } \} \\
& R \cdot S^\circ \subseteq id \\
\equiv & \quad \{ \text{ converses } \} \\
& S \cdot R^\circ \subseteq id \\
\equiv & \quad \{ \text{ law (5.31) } \} \\
& S \cdot \delta\, R \subseteq R
\end{array}
$$

From these we calculate $(R \cup S) \cdot \delta\, R \cdot \delta\, S \subseteq R \cap S$. Since $R \cap S \subseteq (R \cup S) \cdot \delta\, R \cdot \delta\, S$,

$$
\begin{array}{ll}
& R \cap S \subseteq R \cup S \\
\equiv & \quad \{ R \cdot \delta\, R = R \} \\
& (R \cap S) \cdot \delta\, (R \cap S) \subseteq R \cup S \\
\equiv & \quad \{ \text{shunting for correflexive relations} \} \\
& (R \cap S) \cdot \delta\, (R \cap S) \subseteq (R \cup S) \cdot \delta\, (R \cap S) \\
\equiv & \quad \{ R \cdot \delta\, R = R \} \\
& (R \cap S) \subseteq (R \cup S) \cdot \delta\, (R \cap S) \\
\Rightarrow & \quad \{ \delta\, (R \cap S) \subseteq (\delta\, R \cap \delta\, S) \} \\
& (R \cap S) \subseteq (R \cup S) \cdot (\delta\, R \cap \delta\, S)
\end{array}
$$

we obtain $\sqcap = \cap$ for simple relations.

5.6 Structural refinement

We close the technical part of this chapter by showing that the *structural refinement* law

$$S \vdash R \quad \Rightarrow \quad \mathsf{F}\, S \vdash \mathsf{F}\, R \tag{5.32}$$

stems from factorization (5.3). This law expresses $\vdash$-monotonicity of an arbitrary parametric type F. Technically, the parametricity of F is captured by regarding it as a *relator*, see chapter 2.

Fact (5.32) is another example of a property of operation refinement whose proof uses the strategy of promoting $\vdash_{post}/\vdash_{pre}$ properties to $\vdash$. We need the auxiliary result that every $\subseteq$-monotonic relational combinator f which commutes with δ,

$$f \cdot (\mathit{ker}\, \delta) \quad \subseteq \quad (\mathit{ker}\, \delta) \cdot f \tag{5.33}$$

is $\vdash_{post}$-monotonic,

$$f \cdot \vdash_{post} \quad \subseteq \quad \vdash_{post} \cdot f \tag{5.34}$$

and that every relator F is both $\vdash_{pre}/\vdash_{post}$-monotonic:

$$\mathsf{F} \cdot \vdash_{post} \quad \subseteq \quad \vdash_{post} \cdot \mathsf{F} \tag{5.35}$$

$$\mathsf{F} \cdot \vdash_{pre} \quad \subseteq \quad \vdash_{pre} \cdot \mathsf{F} \tag{5.36}$$

The PF-proofs of these results are deferred to appendix B.7. Then the calculation of (5.32) is an easy task:

$$
\begin{array}{cl}
& \text{TRUE} \\
\equiv & \quad \{\ (5.35)\ \} \\
& \mathsf{F} \cdot \vdash_{post} \quad \subseteq \quad \vdash_{post} \cdot \mathsf{F} \\
\Rightarrow & \quad \{\ \text{monotonicity of composition}\ \} \\
& \mathsf{F} \cdot \vdash_{post} \cdot \vdash_{pre} \quad \subseteq \quad \vdash_{post} \cdot \mathsf{F} \cdot \vdash_{pre} \\
\Rightarrow & \quad \{\ (5.36) \text{ and } \subseteq\text{-transitivity}\ \} \\
& \mathsf{F} \cdot \vdash_{post} \cdot \vdash_{pre} \quad \subseteq \quad \vdash_{post} \cdot \vdash_{pre} \cdot \mathsf{F} \\
\equiv & \quad \{\ (5.3)\ \} \\
& \mathsf{F} \cdot \vdash \quad \subseteq \quad \vdash \cdot \mathsf{F} \\
\equiv & \quad \{\ \text{shunt over } \mathsf{F} \text{ (2.15) and then go pointwise on } S \text{ and } R\ \} \\
& R \vdash S \quad \Rightarrow \quad \mathsf{F}\, R \vdash \mathsf{F}\, S
\end{array}
$$

Let us see some examples of the application of law (5.32).

Let $\mathsf{F}\, X = X^\star$, $\mathsf{F}\, R = R^\star$ that is, F is the unary relator associated to finite lists.

This relator has the following definition:

$$
\begin{array}{c}
<a_1, a_2, \cdots, a_n>(R^\star)<b_1, b_2, \cdots, b_m> \\
\equiv \\
n = m \wedge \langle \forall\, i \;:\; 1 \leq i \leq n : \; a_i \, R \, b_i \rangle
\end{array}
$$

Now we can elaborate on F and define

$$
\begin{array}{rcl}
Matrix & = & Row^\star \\
Row & = & Cell^\star \\
Cell & = & \mathbb{R}
\end{array}
$$

Let operation $Sqrt$ be such that it gives the square root of a (positive) real number:

$$
\begin{array}{ll}
Sqrt : & Cell \rightarrow Cell \\
y \; Sqrt \; x \equiv & x \geq 0 \wedge y^2 = x
\end{array}
$$

and functional operation $sqrtf$ which refines $Sqrt$:

$$
\begin{array}{lll}
sqrtf : & Cell \rightarrow Cell & \\
sqrtf \; x \equiv & if & x \geq 0 \\
& then & +\sqrt{x} \\
& else & 0
\end{array}
$$

Of course $Sqrt \vdash sqrtf$ — an example of refinement by increase of definiton and reduction of non-determinism.

As the result of application of functor F two times, we define the following operation over matrixes:

$$MatrixSqrt = (Sqrt^\star)^\star$$

which unfolds into a quite complicated pointwise definition:

$$
\begin{array}{rl}
M' \; MatrixSqrt \; M \equiv & len \; M = len \; M' \wedge \forall i : 1 \leq i \leq \; len \; M : len \; M \; i = len \; M' \; i \wedge \\
& \forall j : 1 \leq j \leq len \; Mi : (M'i)j \; Sqrt \; (Mi)j
\end{array}
$$

By application of law (5.32) two times, we are led to:

$$Sqrt \vdash sqrtf \Rightarrow MatrixSqrt \vdash (sqrtf^\star)^\star$$

Analogously to $MatrixSqrt$ we define $matrixsqrtf$:

$$matrixsqrtf = (sqrtf^\star)^\star$$

which may be defined with a computational flavour as:

$$matrixsqrtf \; M = [[if \; c \geq 0 \; then \; +\sqrt{c} \; else \; 0 \mid c \leftarrow l] \mid l \leftarrow M]$$

In the case of binary relators, $\mathsf{F}\,(A, B)$, law 5.32 means that R and S in $\mathsf{F}\,(R, S)$ can be refined in isolation, say $R \vdash T$ and $S \vdash U$, and the outcome be combined into

$\mathsf{F}\,(T,U)$. Let, for instance $\mathsf{F}\,(A,B) = A\times B$. Law 5.32 can be presented in the following way:

$$\frac{S \vdash R, T \vdash U}{S \times T \vdash R \times U}$$

cf. diagram

$$\begin{tikzcd} A \arrow[d, "S \vdash R"'] & B \arrow[d, "T \vdash U"'] & A \times B \arrow[d, "S\times T\vdash R\times U"'] \\ C & D & C \times D \end{tikzcd}$$

This law makes it possible to refine two separated operations (working on different parts of a state, for instance) in complete isolation, simply by joining the two refinements.

5.7 Data Refinement Revisited

In this section we combine the two orderings involved in software refinement, the one on data structures ($\leq$) studied in the previous chapter and the one ($\vdash$) studied in this chapter.

This combination is referred to in the literature as *data-refinement* and has attracted the attention of many reaserchers, leading to a book [50] on the subject.

Definition 5.1 *Data Refinement in Full [113]*
We introduce simultaneous algorithm/data *refinement as follows: given*

- *a spec* $A \xleftarrow{S} B$
- *abstraction relation* $A \xleftarrow{F_1} C$
- *representation relation* $D \xleftarrow{R_2} B$

then $C \xleftarrow{I} D$ *will be said to* implement S *iff*

$$S \vdash F_1 \cdot I \cdot R_2 \tag{5.37}$$

The situation is pictured in Figure 5.1.

Below we show how relationship (5.37) relates to the standard data refinement notions of *upward* and *downward simulations* [50], in the case of entire relations, in which case $\vdash = \subseteq^\circ$, as we have seen (recall table 5.29).

First of all, we present two definitions extracted from [50]:

Definition 5.2 (Downward simulation) *Assume that* Aop,Cop *are the abstract/concrete operation pair. A downward simulation is a relation* R *from the abstract state space* AS *to the concrete one,* CS*, such that*

$$Cop \cdot R \quad \subseteq \quad R \cdot Aop \tag{5.38}$$

$$\begin{tikzcd}
A \arrow[r, "S"] \arrow[d, "R_2"'] & B \\
C \arrow[r, "I"'] & D \arrow[u, "F_1"']
\end{tikzcd}$$

Figure 5.1: Full Data Refinement

$$\begin{tikzcd}
AS \arrow[d, "R"'] \arrow[dr, phantom, "\supseteq"] & AS \arrow[l, "Aop"'] \arrow[d, "R"] \\
Cs & CS \arrow[l, "Cop"]
\end{tikzcd}$$

Figure 5.2: Downward Simulation

Definition 5.3 (Upward simulation) *For operations Aop,Cop, the abstract and concrete operations, respectivelly, an upward simulation is a relation R from the concrete state space CS to the abstract state space AS, such that*

$$T \cdot Cop \quad \subseteq \quad Aop \cdot T \tag{5.39}$$

Upward Simulation

$$\begin{aligned}
& S \vdash F_1 \cdot I \cdot R_2 \\
\equiv & \qquad \{ \text{ entire} \} \\
& F_1 \cdot I \cdot R_2 \subseteq S \\
\equiv & \qquad \{ \ F_1 = F \text{ and } R_2 = F^{\circ} \} \\
& F \cdot I \cdot F^{\circ} \subseteq S \\
\equiv & \qquad \{ \text{ shunting} \} \\
& F \cdot I \subseteq S \cdot F
\end{aligned}$$

$$\begin{tikzcd}
AS \arrow[dr, phantom, "\subseteq"] & AS \arrow[l, "Aop"'] \\
Cs \arrow[u, "T"] & CS \arrow[l, "Cop"] \arrow[u, "T"']
\end{tikzcd}$$

Figure 5.3: Upward Simulation

Implementation I is a full data refinement of S iff F is an upward simulation which witnesses the refinement of S by I.

Downward Simulation

$$
\begin{array}{cl}
 & S \vdash F_1 \cdot I \cdot R_2 \\
\equiv & \quad \{ \text{entire} \} \\
 & F_1 \cdot I \cdot R_2 \subseteq S \\
\equiv & \quad \{ F_1 = F \text{ and } R_2 = R = F^\circ \} \\
 & F \cdot I \cdot F^\circ \subseteq S \\
\equiv & \quad \{ \text{shunting} \} \\
 & I \cdot F^\circ \subseteq F^\circ \cdot S \\
\equiv & \quad \{ R = F^\circ \} \\
 & I \cdot R \subseteq R \cdot S
\end{array}
$$

Implementation I is a full data refinement of S iff R is a downward simulation which witnesses the refinement of S by I.

We summarize the relationship between full data refinement and standard data refinement as the following equivalence:

$$
S \vdash F \cdot I \cdot R \quad \equiv \quad F \cdot I \subseteq S \cdot F \wedge R = F^\circ \tag{5.40}
$$

which is the result of the following reasoning,

$$
\begin{array}{cl}
 & S \vdash F \cdot I \cdot R \\
\equiv & \quad \{ \text{idempotency} \} \\
 & S \vdash F \cdot I \cdot R \wedge S \vdash F \cdot I \cdot R \\
\equiv & \quad \{ \text{Upward Simulation and Downward Simulation} \} \\
 & F \cdot I \subseteq S \cdot F \wedge I \cdot R \subseteq R \cdot S \\
\equiv & \quad \{ R = F^\circ \} \\
 & F \cdot I \subseteq S \cdot F \wedge I \cdot F^\circ \subseteq F^\circ \cdot S \\
\equiv & \quad \{ \text{shunting} \} \\
 & F \cdot I \subseteq S \cdot F \wedge F \cdot I \subseteq S \cdot F \\
\equiv & \quad \{ \text{idempotency} \} \\
 & F \cdot I \subseteq S \cdot F
\end{array}
$$

Discussion. In practice, operations are very often total (ie. entire) and we just need downward simulations to prove refinement, in most cases [32]. This is the case where operations are functions. Full data refinement is equivalent to downward simulation if we assume that abstraction relation is entire (thus a function) and the representation relation is its converse. These assumptions are acceptable, as we shall see in the example of the next section.

5.8 Example of Data Refinement

5.8.1 Specification and Implementation of a Bounded Buffer in VDM

Below we take as example the refinement of the operations on a bounded buffer data type $Buffer$ which can be found in [142] specified in the Z notation. We shall calculate a functional implementation based on a concrete type $Array$. These two types will be related by a retrieve function $retriveBuffer$ which is surjective on valid states.

We start by presenting the abstract model in VDM-SL notation:

```
-- Abstract Model
types

X = token;

Buffer :: buffer: seq of X
          maxsize : nat
          inv b == len b.buffer <= b.maxsize;

Output = Ok | <full> | <empty>;
Ok :: x:X;
rBuffer = Buffer*Output;

functions

BufferInit() r:Buffer
pre true
post r=mk_Buffer([],1000);

BufferIn(b:Buffer,x:X) r: rBuffer
pre true
post (len b.buffer<b.maxsize and
      r= mk_(mk_Buffer(b.buffer^[x],b.maxsize), mk_Ok(x))
     ) or
     (len b.buffer=b.maxsize and
      r= mk_(b,<full>));

BufferOut(b:Buffer) r: rBuffer
pre true
post ( b.buffer <> [] and
       r= mk_(mk_Buffer(tl b.buffer, b.maxsize),
              mk_Ok(hd b.buffer))
     ) or
     ( b.buffer = [] and
```

```
          r= mk_(b,<empty>));
```

It can be easily checked that the operations over $Buffer$ preserve the abstract invariant and that the initialization operation $BufferInit$ induces a valid state.

The proposed concrete model follows:

```
types

Array :: array: seq of X
         maxsize: nat
         bot: nat
         top:nat
         size:nat
         inv a == a.bot>=1 and a.bot<=a.maxsize and
                  a.top>=1 and a.top<=a.maxsize and
                  a.size>=0 and a.size<=a.maxsize and
                  a.size mod a.maxsize =
                            (a.top-(a.bot-1)) mod a.maxsize;

functions

arrayInit: () -> Array
arrayInit() ==
mk_Array([],1000,1,1000,0);

arrayIn: Array*X -> Array*Output
arrayIn(a,x) ==
if a.size<a.maxsize
then let
       y=(a.top mod a.maxsize)+1
     in
       mk_(mk_Array(a.array++{y|->x},
           a.maxsize,
           a.bot,
           y,
           a.size+1),
           mk_Ok(x))
else mk_(a,
         <full>);

arrayOut: Array -> Array*Output
ArrayOut(a) ==
if a.size <> 0
then mk_(mk_Array(a.array,
    a.maxsize,
    (a.bot mod a.maxsize)+1,
```

```
        a.top,
        a.size-1),
        mk_Ok(a.array(a.bot)))
    else mk_(a, <empty>);
```

Data-level refinement is based on retrieve function

```
retrieveBuffer: Array -> Buffer
retrieveBuffer(a)==
     mk_Buffer(extract(a.size,shift(a.bot-1,a.array)), a.maxsize);

shift: nat*seq of X -> seq of X
shift(n,s) ==
     if n=0
     then s
     elseif s= []
     then []
     else shift(n-1,tl s ^ [hd s]);

extract: nat*seq of X -> seq of X
extract(n,s) == if n=0 then []
                elseif s= [] then []
                else [hd s]^extract(n-1,tl s);
```

which is surjective, cf. [142].

5.8.2 Correctness of the Implementation

We shall prove that function $arrayIn$ of the implementation follows from relational operation $BufferIn$ of the specification. For the remaining operations the reasoning is similar.

We start by observing that the post-condition of operation $BufferIn$ has pattern

$$BufferIn = f \cdot [\![p]\!] \cup g \cdot [\![p']\!] \tag{5.41}$$

for the obvious definition of p, f, p' and g:

$$\begin{aligned}
p &= \mathit{len}\, b.buffer < b.maxsize \\
f(b,x) &= mk(mkBuffer(b.buffer \frown x, b.maxsize), mkOk(x))) \\
p' &= \mathit{len}\, b.buffer = b.maxsize \\
g(b,x) &= mk(b, < \mathit{empty} >)
\end{aligned}$$

We calculate:

$$
\begin{array}{ll}
 & BufferIn \\
= & \quad \{ \text{ (5.41) assuming definition above} \} \\
 & f \cdot [\![p]\!] \cup g \cdot [\![p']\!] \\
\subseteq & \quad \{ \; p' \subseteq id - [\![p]\!] \text{ and monotonicity} \} \\
 & f \cdot [\![p]\!] \cup g \cdot (id - [\![p]\!]) \\
= & \quad \{ \text{ (B.24, B.25) } \} \\
 & f \cdot i_1^\circ \cdot p? \cup g \cdot i_2^\circ \cdot p? \\
= & \quad \{ \; (\cdot X) \text{ is a lower adjoint} \} \\
 & (f \cdot i_1^\circ \cup g \cdot i_2^\circ) \cdot p? \\
= & \quad \{ \text{ definition of either} \} \\
 & [f, g] \cdot p? \\
= & \quad \{ \text{ definition of McCarthy conditional} \} \\
 & [\![p]\!] \rightarrow f, g \\
= & \quad \{ \text{ by definition} \} \\
 & bufferIn'
\end{array}
$$

Thus we have reached explicit function

```
bufferIn': Buffer*X -> Buffer*Output
bufferIn'(b,x) ==
    if len b.buffer < b.maxsize
    then mk_(mk_Buffer(b.buffer^[x],b.maxsize), mk_Ok(x))
    else mk_(b, <full>);
```

Altogether,

$$BufferIn \subseteq bufferIn'$$

This furthermore shows that $BufferIn$ is simple, since *smaller than simple is simple*. Thanks to table (5.29) we can safely establish

$$BufferIn \;\; \vdash \;\; bufferIn' \tag{5.42}$$

Proceeding with the refinement process, we now need to show that full data refinement

$$bufferIn' \;\; \vdash_{post} \;\; (rb \times id) \cdot arrayIn \cdot (rb \times id)^\circ \tag{5.43}$$

holds. This is a $\vdash_{post}$-refinement because all relations involved are entire [2], cf. table (5.29). We proceed:

$$
\begin{array}{cl}
& bufferIn' \vdash_{post} (rb \times id) \cdot arrayIn \cdot (rb \times id)^{\circ} \\
\equiv & \qquad \{ \text{ (5.29)} \} \\
& (rb \times id) \cdot arrayIn \cdot (rb \times id)^{\circ} \subseteq bufferIn' \\
\equiv & \qquad \{ \text{ shunting } \} \\
& (rb \times id) \cdot arrayIn \subseteq bufferIn' \cdot (rb \times id) \\
\equiv & \qquad \{ \text{ equality of functions } \} \\
& (rb \times id) \cdot arrayIn = bufferIn' \cdot (rb \times id)
\end{array}
$$

cf. diagram

$$
\begin{tikzcd}
Buffer \times Output \arrow[r, "bufferIn'"] & Buffer \times X \\
Array \times Output \arrow[u, "rb \times id"] \arrow[r, "arrayIn"'] & Array \times X \arrow[u, "rb \times id"']
\end{tikzcd}
$$

From this equation, we are going to calculate the functional solution $arrayIn$, the unknown of the equation. We reason:

$$
\begin{array}{ll}
 & (rb \times id) \cdot arrayIn = bufferIn' \cdot (rb \times id) \\
\equiv & \quad \{ \text{ let } bufferIn' = \langle h, j \rangle \text{ and } arrayIn = \langle f, g \rangle \} \\
 & (rb \times id) \cdot \langle f, g \rangle = \langle h, j \rangle \cdot (rb \times id) \\
\equiv & \quad \{ \times\text{-absorption and } \times\text{-fusion} \} \\
 & \langle rb \cdot f, g \rangle = \langle h \cdot (rb \times id), j \cdot (rb \times id) \rangle \\
\Leftarrow & \quad \{ \text{ equalities from structured equality} \} \\
 & rb \cdot f = h \cdot (rb \times id) \wedge g = j \cdot (rb \times id) \\
\equiv & \quad \{ \text{ let } \langle h, j \rangle = p \rightarrow k \ , \ l \text{ and cancellation} \} \\
 & rb \cdot f = \pi_1 \cdot (p \rightarrow k \ , \ l) \cdot (rb \times id) \wedge g = \pi_2 \cdot (p \rightarrow k \ , \ l) \cdot (rb \times id) \\
\equiv & \quad \{ \ f \cdot (p \rightarrow g \ , \ h) = p \rightarrow (f \cdot g) \ , \ (f \cdot h) \ \} \\
 & rb \cdot f = (p \rightarrow \pi_1 \cdot k \ , \ \pi_1 \cdot l) \cdot (rb \times id) \wedge g = (p \rightarrow \pi_2 \cdot k \ , \ \pi_2 \cdot l) \cdot (rb \times id) \\
\equiv & \quad \{ \ (p \rightarrow f \ , \ g) \cdot h = (p \cdot h) \rightarrow (f \cdot h) \ , \ (g \cdot h) \ \} \\
 & rb \cdot f = (p \cdot (rb \times id) \rightarrow \pi_1 \cdot k \cdot (rb \times id) \ , \ \pi_1 \cdot l \cdot (rb \times id) \wedge \\
 & g = (p \cdot (rb \times id) \rightarrow \pi_2 \cdot k \cdot (rb \times id) \ , \ \pi_2 \cdot l \cdot (rb \times id)) \\
\equiv & \quad \{ \text{ going pointwise} \} \\
 & rb(f(A, x)) = (p(rb\ A, x) \rightarrow \pi_1(k(rb\ A, x)) \ , \ \pi_1(l(rb\ A, x)) \\
 & g(A, x) = (p(rb\ A, x) \rightarrow mk\text{Ok}(x) \ , \ FULL) \\
\equiv & \quad \{ \text{ going pointwise continued} \} \\
 & rb(f(A, x)) = \ (len(rb\ A).buffer < (rb\ A).\text{maxsize}) \rightarrow \\
 & \qquad mk\text{Buffer}((rb\ A).buffer \frown [x], (rb\ A).\text{maxsize}), \\
 & \qquad rb\ A) \\
 & g(A, x) = (p(rb\ A, x) \rightarrow mk\text{Ok}(x) \ , \ FULL) \\
\equiv & \quad \{ \text{ justifications below and } f\ (p \rightarrow x \ , \ y) = p \rightarrow f\ x \ , \ f\ y \} \\
 & let \quad y = (A.top\ mod\ A.\text{maxsize}) + 1 \\
 & in \quad rb(f(A, x)) = \ rb\ (A.size < A.\text{maxsize}) \rightarrow \\
 & \qquad mk\text{Array}(A.array + +[y \mapsto x], A.\text{maxsize}, A.bot, y, A.size + 1), \\
 & \qquad A) \\
 & g(A, x) = (A.size < A.\text{maxsize} \rightarrow mk\text{Ok}(x) \ , \ FULL) \\
\equiv & \quad \{ \text{ solving the equation for } f \} \\
 & f(A, x) \ = \ let \quad y = (A.top\ mod\ A.\text{maxsize}) + 1 \\
 & \qquad in \quad A.size < A.\text{maxsize}) \rightarrow \\
 & \qquad mk\text{Array}(A.array + +[y- > x], A.\text{maxsize}, A.bot, y, A.size + 1), \\
 & \qquad A \\
 & g(A, x) \ = \ (A.size < A.\text{maxsize} \rightarrow mk\text{Ok}(x) \ , \ FULL)
\end{array}
$$

We can calculate the final predicate corresponding to the McCarthy conditional as follows,

$$\begin{array}{ll} & len(rb\ A).buffer < (rb\ A).maxsize \\ \equiv & \quad \{\ len(rb\ A).buffer = A.size \text{ and } (rb\ A).maxsize = A.maxsize\} \\ & A.size < A.maxsize \end{array}$$

by definition of rb, in particular of $extract$.

The following equality [142] is assumed in the above proof:

$$\begin{array}{ll} & mk\text{Buffer}((rb\ A).buffer \frown [x], (rb\ A).maxsize) = \\ let & y = (A.top\ mod\ A.maxsize) + 1 \\ in & rb(mk\text{Array}(A.array + +[y \mapsto x], A.maxsize, A.bot, y, A.size + 1)) \end{array}$$

Finally, we can define the operation $arrayIn$ from the calculations made before, as

$$arrayIn(A, X) = \langle f, g\rangle(A, x) = (f(A, x), g(A, x))$$

which is equivalent to the definition presented in the concrete model:

```
arrayIn: Array*X -> Array*Output
arrayIn(a,x) ==
if a.size<a.maxsize
then let
       y=(a.top mod a.maxsize)+1
     in
       mk_(mk_Array(a.array++{y|->x},a.maxsize,a.bot,y,a.size+1),
           mk_Ok(x))
else mk_(a,
         <full>);
```

We are almost done in proving that the initial relational operation $BufferIn$ is fully data refined by $arrayIn$: from (5.42) and (5.43), we draw by transitivity

$$BufferIn \ \ \vdash \ \ (rb \times id) \cdot arrayIn \cdot (rb \times id)^{\circ} \tag{5.44}$$

the statement of the full data refinement of operation $BufferIn$ by $arrayIn$, as required.

5.9 Summary

Refinement is among the most studied topics in software design theory. An extensive treatment of the subject can be found in [50]. It is, however, far from being an easy-to-use body of knowledge, a remark which is mirrored on terminology — cf. *downward*, *upward* refinement [64], *forwards*, *backwards* refinement [64, 142, 87], etc.

Boudriga *et al* [35] refer prosaically to the refinement ordering (denoted $\vdash$ in the current chapter) as the *less defined ordering* on pre/post-specifications. "Less defined" has a double meaning in this context: smaller domain-wise and vaguer range-wise. But such a linguistic consensus is not found in the underlying mathematics: $\vdash$ merges two opposite orderings, one pushing towards smaller specs and another to larger ones.

With the purpose to better understand this opposition, we decided to invest on a factorization which was proposed by Lindsay Groves in [61] but left exploited henceforth (to the best of the author's knowledge). Our approach to this factorization, which is calculational and *pointfree*-relational, contrasts with the hybrid models usually found in the literature, which typically use relational combinators to express definitions and properties but perform most reasoning steps at point-level, eg. using set-valued functions.

Chapter 6

Coalgebraic Refinement

6.1 Introduction

Suppose one needs to replace part of a system by another? How safe is such a replacement? Milner [88] answered this question long ago by requiring that they should be *bisimilar*. In [88] a non-deterministic specification is implemented by a concurrent system, and the link between these is bisimilarity.

Informally, one can say that two systems are bisimilar if they behave in the same way, i.e. an observer can't distinguish them. Bisimilarity involves the concept of bisimulation, nowadays discussed in the broader context of coalgebras [132, 4], which generalize (labelled) transition systems (LTS).

An LTS is simply a transition relation $Act \times P \xleftarrow{R} P$ involving a set of *processes* P and a set of *observations* or *actions* Act. Quite often R is transposed to a function $\overline{R}$ of type $(P \longleftarrow P)^{Act}$ so that, for all $\alpha \in Act$,

$$q(\overline{R}\,\alpha)p \quad \equiv \quad (\alpha, q)\; R\; p$$

holds. Wherever R is implicit in the context, notation $q \overset{\alpha}{\leftarrow} p$ (meaning *q is reachable from p via action or observation α*) will abbreviate $q(\overline{R}\,\alpha)p$. The converse of this relation is $p \overset{\alpha}{\rightarrow} q$, meaning *observation α happens or is observable wherever p evolves to q*. So $\overset{\alpha}{\rightarrow} \;=\; (\overset{\alpha}{\leftarrow})^{\circ}$.

A coalgebra is a generalized transition system. Given an endofunctor F over Set a F-coalgebra or F-system is a function $\mathsf{F}\,S \xleftarrow{\alpha_S} S$, for F an endofunctor over Set. Set S is referred to as *carrier* of the system, also understood as a set of *states*; function α_S is the F-transition structure (or dynamics) of the system.

Clearly, the power-transpose of a given LTS R, $\mathcal{P}((A \times P)) \xleftarrow{\Lambda R} P$, is a B-coalgebra (or B-system) for

$$\mathsf{B}\,X = \mathcal{P}(A \times X)$$

The concept of coalgebra is dual to that of an algebra. However, universal coalgebra is not simply the dual theory corresponding to universal algebra. Coalgebras are related by homomorphisms, as defined below, which involve refinement concepts.

Definition 6.1 *Let $\langle X, p : X \longrightarrow \mathsf{F}\,X\rangle$ and $\langle Y, q : Y \longrightarrow \mathsf{F}\,Y\rangle$ be* coalgebras *for functor* F. *A* morphism *connecting p and q is a function h between their carriers such that the following diagram commutes:*

$$\begin{array}{ccc} X & \xrightarrow{p} & \mathsf{F}\,X \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,h} \\ Y & \xrightarrow[q]{} & \mathsf{F}\,Y \end{array} \qquad \textit{i.e.} \qquad q \cdot h = \mathsf{F}\,h \cdot p \tag{6.1}$$

In this chapter we approach coalgebraic refinement. As we shall see, this can be done via relational refinement of the corresponding transition relations. A very important concept in the theory of both LTS and coalgebraic systems is that of *bisimulation* [88]. In this chapter we integrate bisimulation in the broader area of coalgebraic refinement.

6.1.1 Strong Bisimulation

Definition 6.2 *Given LTS R as above, a binary relation $P \xleftarrow{S} P$ is said to be a* strong bisimulation *iff, for all α in Act, $q\ S\ p$ — that is, $p\ S\ q$ — implies*

(i) *Whenever $p \xrightarrow{\alpha} p'$, then, for some q', $q \xrightarrow{\alpha} q'$ and $(p', q') \in S$*

(ii) *Whenever $q \xrightarrow{\alpha} q'$, then, for some p', $p \xrightarrow{\alpha} p'$ and $(p', q') \in S$*

As before, our approach to handling and reasoning about this definition will start from PF-transforming it. Our starting point is, therefore, a rather long-winded formula:

$$\left\langle \begin{array}{c} \forall\, p, q\ : \\ p\ S\ q : \\ \langle \forall\, p'\ :\ p \xrightarrow{\alpha} p' :\ \langle \exists\, q'\ ::\ q \xrightarrow{\alpha} q' \wedge (p', q') \in S \rangle\rangle \\ \wedge \\ \langle \forall\, q'\ :\ q \xrightarrow{\alpha} q' :\ \langle \exists\, p'\ ::\ p \xrightarrow{\alpha} p' \wedge (p', q') \in S \rangle\rangle \end{array} \right\rangle$$

Let us first remove the existential quantifiers via law for composition in table (2.32) (twice):

$$\langle \forall\, p, q\ :\ p\ S\ q :\ \langle \forall\, p'\ :\ p' \xleftarrow{\alpha} p :\ p'(S \cdot \xleftarrow{\alpha})q \rangle \wedge \langle \forall\, q'\ :\ q' \xleftarrow{\alpha} q :\ q'(S^\circ \cdot \xleftarrow{\alpha})p \rangle\rangle$$

Then we can do the same to the inner universal ones, this time via law for right division in table (2.32):

$$\langle \forall\, p, q\ :\ p\ S\ q :\ p(\xleftarrow{\alpha} \backslash (S \cdot \xleftarrow{\alpha}))q \wedge q(\xleftarrow{\alpha} \backslash (S^\circ \cdot \xleftarrow{\alpha}))p \rangle$$

We are left with a sole universal quantifier which commutes with the inner conjunction,

$$\langle \forall\, p, q\ :\ p\ S\ q :\ p(\xleftarrow{\alpha} \backslash (S \cdot \xleftarrow{\alpha}))q \rangle \ \wedge\ \langle \forall\, p, q\ :\ p\ S\ q :\ q(\xleftarrow{\alpha} \backslash (S^\circ \cdot \xleftarrow{\alpha}))p \rangle$$

leading to two pointfree inclusions via rule for inclusion in table (2.32) and converse:

$$S \subseteq \overset{\alpha}{\leftarrow} \backslash (S \cdot \overset{\alpha}{\leftarrow}) \;\wedge\; S^\circ \subseteq \overset{\alpha}{\leftarrow} \backslash (S^\circ \cdot \overset{\alpha}{\leftarrow})$$

These two inclusions lead finally to

$$\overset{\alpha}{\leftarrow} \cdot S \subseteq S \cdot \overset{\alpha}{\leftarrow} \;\;\wedge\;\; \overset{\alpha}{\leftarrow} \cdot S^\circ \subseteq S^\circ \cdot \overset{\alpha}{\leftarrow} \tag{6.2}$$

thanks to the Galois connection that relates division with composition. While thus far we have regarded S as an endo-relation on P, it can be generalized to a relation $P' \xleftarrow{S} P$ involving the process sets P and P' of two different LTS R and R', leading to the same formula on a wider type:

$$\overset{\alpha}{\leftarrow} \cdot S \subseteq S \cdot \overset{\alpha}{\leftarrow} \;\wedge\; \overset{\alpha}{\leftarrow} \cdot S^\circ \subseteq S^\circ \cdot \overset{\alpha}{\leftarrow} \quad \text{cf. diagram} \quad \begin{tikzcd} P' & P' \arrow[l, "\overset{\alpha}{\leftarrow}"'] \\ P \arrow[u, "S"] \arrow[ur, phantom, "\supseteq"] & P \arrow[l, "\overset{\alpha}{\leftarrow}"] \arrow[u, "S"'] \end{tikzcd} \tag{6.3}$$

This is what is meant by *bisimulation* in the LTS literature, see eg. [119, 88], the qualifier *strong* being added to mean a bisimulation of an LTS by itself. Note that both conjuncts of (6.3) "share the same pattern", that of what is referred to as *simulation* in the literature. So a bisimulation S is a simulation such that its converse S° is also a simulation.

The PF-definition which we were led to above is far more amenable to calculation than the original pointwise wherefrom we started. For example, the identity emerges trivially as a bisimulation, and composition of two bisimulations $P' \xleftarrow{S} P$ and $P'' \xleftarrow{T} P'$ is a bisimulation (the calculation concerning the $(T \cdot S)^\circ$ simulation is analogous):

$$\begin{array}{cl}
& \overset{\alpha}{\leftarrow} \cdot T \cdot S \subseteq T \cdot S \cdot \overset{\alpha}{\leftarrow} \\
\Leftarrow & \quad \{ \text{ since } \overset{\alpha}{\leftarrow}{}' \cdot S \subseteq S \cdot \overset{\alpha}{\leftarrow} \text{ and thus } T \cdot \overset{\alpha}{\leftarrow}{}' \cdot S \subseteq T \cdot S \cdot \overset{\alpha}{\leftarrow} \ \} \\
& \overset{\alpha}{\leftarrow} \cdot T \cdot S \subseteq T \cdot \overset{\alpha}{\leftarrow}{}' \cdot S \\
\Leftarrow & \quad \{ \text{ monotonicity of } (\cdot S) \ \} \\
& \overset{\alpha}{\leftarrow} \cdot T \subseteq T \cdot \overset{\alpha}{\leftarrow}{}' \\
\equiv & \quad \{ \text{ simulation } T \text{ is assumed } \} \\
& \text{TRUE}
\end{array}$$

The proof that bisimulations are closed under relational union is also a manageable one:

$$
\begin{array}{cl}
& \overset{\alpha}{\leftarrow}' \cdot \langle \bigcup\, i\ ::\ S_i \rangle \subseteq \langle \bigcup\, i\ ::\ S_i \rangle \cdot \overset{\alpha}{\leftarrow} \\
\equiv & \qquad \{\ (\cdot S) \text{ and } (S\cdot) \text{ are lower adjoints} \} \\
& \langle \bigcup\, i\ ::\ \overset{\alpha}{\leftarrow}' \cdot S_i \rangle \subseteq \langle \bigcup\, i\ ::\ S_i \cdot \overset{\alpha}{\leftarrow} \rangle \\
\equiv & \qquad \{\ \text{universal property of join} \} \\
& \overset{\alpha}{\leftarrow}' \cdot S_i \subseteq \langle \bigcup\, i\ ::\ S_i \cdot \overset{\alpha}{\leftarrow} \rangle \\
\Leftarrow & \qquad \{\ \text{each } S_i \text{ is a simulation; monotonicity and transitivity } \} \\
& \overset{\alpha}{\leftarrow}' \cdot S_i \subseteq \langle \bigcup\, i\ ::\ \overset{\alpha}{\leftarrow}' \cdot S_i \rangle \\
\equiv & \qquad \{\ R_1 \subseteq R_1 \cup R_2 \text{ for the } n\text{-ary case} \} \\
& \textsc{True}
\end{array}
$$

Thus the converse of a bisimulation $P' \xleftarrow{S} P$ is also a bisimulation $P \xleftarrow{S^\circ} P'$, the former being expressed as

$$\overset{\alpha}{\leftarrow} \cdot S \subseteq S \cdot \overset{\alpha}{\leftarrow} \ \wedge\ \overset{\alpha}{\leftarrow} \cdot S^\circ \subseteq S^\circ \cdot \overset{\alpha}{\leftarrow}$$

the latter one as

$$\overset{\alpha}{\leftarrow} \cdot S^\circ \subseteq S^\circ \cdot \overset{\alpha}{\leftarrow} \ \wedge\ \overset{\alpha}{\leftarrow} \cdot (S^\circ)^\circ \subseteq (S^\circ)^\circ \cdot \overset{\alpha}{\leftarrow}$$

The two previous expressions are equivalent since converse is an involution (2.4).

In summary, a strong bisimulation S is a binary endo-relation such that its converse and itself are *simulations*, according to the following (more general) definition of simulation.

Definition 6.3 *A simulation $P' \xleftarrow{S} P$ between two LTS R' and R with the same action set Act is a relation such that, for all $\alpha \in Act$,*

$$\overset{\alpha}{\leftarrow}' \cdot S \subseteq S \cdot \overset{\alpha}{\leftarrow} \qquad
\begin{tikzcd}
P' \arrow[from=r, "\overset{\alpha}{\leftarrow}'"'] & P' \\
P \arrow[u, "S"] \arrow[ur, phantom, "\supseteq"] & P \arrow[l, "\overset{\alpha}{\leftarrow}"] \arrow[u, "S"']
\end{tikzcd} \tag{6.4}$$

6.2 Bisimulation iff Homomorphism

An important lemma in [15] expresses bisimulations in allegorical terms and proves this equivalent to Aczel and Mendler's categorical definition, cf. [3].

Definition 6.4 (Bisimulation for Dialgebras) *Suppose $k : \mathsf{G}A \rightarrow \mathsf{F}\,A$ is a (*F*,*G*)-dialgebra. A relation R of type $A \leftarrow A$ is a $bisimulation$ of k if*

$$\mathsf{G}R \subseteq k^{\circ} \cdot \mathsf{F}\,R \cdot k$$

This definition introduces the concept of an (F,G)-dialgebra for F, G relators, which generalizes both that of algebra and coalgebra.

An F-algebra is clearly an (Id,F)-dialgebra and an F-coalgebra is an (F,Id)-dialgebra, where Id denotes the identity functor.

We translate the previous definition to coalgebras, ie.

Definition 6.5 (Bisimulation for Coalgebras) *Suppose $k : A \rightarrow \mathsf{F}A$ is a* F*-coalgebra. A relation R of type $A \leftarrow A$ is a $bisimulation$ of k if*

$$R \subseteq k^{\circ} \cdot \mathsf{F}R \cdot k$$

This definition introduces a generalized concept, viz. an F-coalgebra for F a relator.

Lemma 6.1 *Let k be an* F*-coalgebra. Function h is a coalgebra (endo)homomorphism iff it is a bisimulation of k.*
Proof:

$$\begin{array}{ll} & h \subseteq k^{\circ} \cdot \mathsf{F}\,h \cdot k \\ \equiv & \quad \{ \textit{ shunting (2.15)} \} \\ & k \cdot h \subseteq \mathsf{F}\,h \cdot k \\ \equiv & \quad \{\, f = g \equiv f \subseteq g \text{ (2.21)} \} \\ & k \cdot h = \mathsf{F}\,h \cdot k \end{array}$$

So a functional bisimulation over a given coalgebra is an homomorphism, and conversely, an (endo)homomorphism is a functional bisimulation over the involved coalgebra.

6.3 Coalgebraic Refinement iff Data Refinement

In the context of coalgebraic refinement, we introduce the following definition, cf. [21].

Definition 6.6 (Forward (backward) morphism) *Let* T *be an extended polynomial functor on Set and consider two* T*-coalgebras $\beta : \mathsf{T}V \longleftarrow V$ and $\alpha : \mathsf{T}U \longleftarrow U$.*

A forward morphism $h : \alpha \longleftarrow \beta$ with respect to a preorder $\leq$, is a function from V to U such that

$$\mathsf{T}h \cdot \beta \,\dot{\leq}\, \alpha \cdot h$$

where $\dot{\leq}$ is the pointwise lifting of $\leq$ given by (2.10).
Dually, h is said to be a backward morphism if

$$\alpha \cdot h \,\dot{\leq}\, \mathsf{T}h \cdot \beta$$

The two refinement concepts are parameterized by preorder $\leq$, which is characterized in [21] and has as particular instance the generic inclusion associated to functor T*.*

In the context of this definition,

- forward morphism h preserves the transition relation $\overset{\beta}{\longleftarrow} = \epsilon \cdot \beta$ corresponding to coalgebra β,

$$v' \overset{\beta}{\longleftarrow} v \Rightarrow hv' \overset{\alpha}{\longleftarrow} hv$$

- backward morphism h reflects the transition relation $\overset{\alpha}{\longleftarrow} = \epsilon \cdot \alpha$ corresponding to coalgebra α,

$$u' \overset{\alpha}{\longleftarrow} hv \Rightarrow \langle \exists\, v' \ : \ v' \in V : \ v' \overset{\beta}{\longleftarrow} v \wedge u' = hv' \rangle$$

Recall from chapter 5 that refinement was discussed in terms of two notions: downward simulation (5.38) and upward simulation (5.39). Our first concern is to relate such notions with foward and backward morphisms (just introduced). The relationship is easily established in the case of functional simulations. We will see that upward (respectively downward) functional simulations correspond to forward (resp. backward) morphisms.

In this way we show the close relationship between coalgebraic refinement [21] and relational refinement [32].

Lemma 6.2 (Upward simulation and forward morphism) *Let $R := h$ in the definition of an upward simulation, recall (5.39):*

$$h \cdot Cop \subseteq Aop \cdot h \tag{6.5}$$

Let T be the functor associated to relational transpose Γ_{T} (recall defiinition 3.1). Then (6.5) is equivalent to forward morphism condition

$$\mathsf{T}h \cdot \Gamma_{\mathsf{T}} Cop \,\dot{\leq}\, (\Gamma_{\mathsf{T}} Aop) \cdot h$$

where $\leq$ is the generic inclusion *relation associated to the transpose (3.14).*

Proof:

$$
\begin{array}{ll}
 & h \cdot Cop \subseteq Aop \cdot h \\
\equiv & \quad \{ \textit{ cancellation (3.9) } \} \\
 & \epsilon_{\mathsf{T}} \cdot \Gamma_{\mathsf{T}}(h \cdot Cop) \subseteq \epsilon_{\mathsf{T}} \cdot \Gamma_{\mathsf{T}}(Aop \cdot h) \\
\equiv & \quad \{ \textit{ Galois connection for division in (2.15)} \} \\
 & \Gamma_{\mathsf{T}}(h \cdot Cop) \subseteq \epsilon_{\mathsf{T}} \backslash (\epsilon_{\mathsf{T}} \cdot \Gamma_{\mathsf{T}}(Aop \cdot h)) \\
\equiv & \quad \{ R\backslash(S \cdot f) = (R\backslash S) \cdot f \} \\
 & \Gamma_{\mathsf{T}}(h \cdot Cop) \subseteq (\epsilon_{\mathsf{T}} \backslash \epsilon_{\mathsf{T}}) \cdot (\Gamma_{\mathsf{T}} Aop \cdot h) \\
\equiv & \quad \{ \textit{ define } \leq = (\epsilon_{\mathsf{T}} \backslash \epsilon_{\mathsf{T}}) \} \\
 & \Gamma_{\mathsf{T}}(h \cdot Cop) \stackrel{.}{\leq} (\Gamma_{\mathsf{T}} Aop \cdot h) \\
\equiv & \quad \{ \textit{ fusion (3.11), since } \Gamma_{\mathsf{T}} Aop \cdot h \textit{ is a function } \} \\
 & \Gamma_{\mathsf{T}}(h \cdot Cop) \stackrel{.}{\leq} (\Gamma_{\mathsf{T}} Aop) \cdot h \\
\equiv & \quad \{ \textit{ absorption property (3.13) with side condition equivalent to } True \} \\
 & (\mathsf{T}h) \cdot \Gamma_{\mathsf{T}} Cop \stackrel{.}{\leq} (\Gamma_{\mathsf{T}} Aop) \cdot h
\end{array}
$$

The side condition of the absorption property is justified by the naturality condition (3.40)

$$h \cdot \epsilon_{\mathsf{T}} = \epsilon_{\mathsf{T}} \cdot \mathsf{T}h$$

By a similar reasoning we state the lemma which follows.

Lemma 6.3 (Downward simulation and backward morphism) *Let $R := h$ in the definition of a downward simulation, recall (5.38):*

$$Cop \cdot h \subseteq h \cdot Aop \tag{6.6}$$

Let T be the functor associated to relational transpose Γ_{T}. Then (6.6) is equivalent to forward morphism condition

$$(\Gamma_{\mathsf{T}} Cop) \cdot h \stackrel{.}{\leq} \mathsf{T}h \cdot (\Gamma_{\mathsf{T}} Aop)$$

where $\leq$ is the generic inclusion *relation associated to the transpose (3.14).*

Proof:

$$
\begin{array}{ll}
 & Cop \cdot h \subseteq h \cdot Aop \\
\equiv & \qquad \{ \textit{ cancellation (3.9) } \} \\
 & \epsilon_{\mathsf{T}} \cdot \Gamma_{\mathsf{T}}(Cop \cdot h) \subseteq \epsilon_{\mathsf{T}} \cdot \Gamma_{\mathsf{T}}(h \cdot Aop) \\
\equiv & \qquad \{ \textit{ galois connection for division in (2.15)} \} \\
 & \Gamma_{\mathsf{T}}(Cop \cdot h) \subseteq \epsilon_{\mathsf{T}} \backslash (\epsilon_{\mathsf{T}} \cdot \Gamma_{\mathsf{T}}(h \cdot Aop)) \\
\equiv & \qquad \{ \; R \backslash (S \cdot f) = (R \backslash S) \cdot f \} \\
 & \Gamma_{\mathsf{T}}(Cop \cdot h) \subseteq (\epsilon_{\mathsf{T}} \backslash \epsilon_{\mathsf{T}}) \cdot \Gamma_{\mathsf{T}}(h \cdot Aop) \\
\equiv & \qquad \{ \textit{ let } \leq = \epsilon_{\mathsf{T}} \backslash \epsilon_{\mathsf{T}} \} \\
 & \Gamma_{\mathsf{T}}(Cop \cdot h) \mathrel{\dot{\leq}} \Gamma_{\mathsf{T}}(h \cdot Aop) \\
\equiv & \qquad \{ \textit{ fusion (3.11), since } \Gamma_{\mathsf{T}} Cop \cdot h \textit{ is a function } \} \\
 & \Gamma_{\mathsf{T}} Cop \cdot h \mathrel{\dot{\leq}} \Gamma_{\mathsf{T}}(h \cdot Aop) \\
\equiv & \qquad \{ \textit{ absorption property (3.13) with side condition equivalent to } True \} \\
 & \Gamma_{\mathsf{T}} Cop \cdot h \mathrel{\dot{\leq}} \mathsf{T} h \cdot \Gamma_{\mathsf{T}} Aop
\end{array}
$$

These two lemmas show a close relationship among coalgebras whose functor support transposition and their relational counterparts. In fact, for any such coalgebra α, its transition relation is easy to spell out:

$$\xleftarrow{\alpha} \stackrel{\mathrm{def}}{=} \epsilon_{\mathsf{T}} \cdot \alpha \tag{6.7}$$

In this context, a homomorphism as defined in (6.1) has the following alternative definition, cf. [21]:

Definition 6.7 *Let* $\langle X, p : X \longrightarrow \mathsf{F}\, X \rangle$ *and* $\langle Y, q : Y \longrightarrow \mathsf{F}\, Y \rangle$ *be* coalgebras *for functor* F. *A* morphism *connecting* p *and* q *is a function* h *between their carriers such that*

$$h \cdot \xrightarrow{p} \;=\; \xrightarrow{q} \cdot h \tag{6.8}$$

cf. the following diagram:

$$
\begin{tikzcd}
X \arrow[r, "h"] \arrow[d, "\xrightarrow{p}"'] & Y \arrow[d, "\xrightarrow{q}"] \\
X \arrow[r, "h"'] & Y
\end{tikzcd}
$$

Moreover, we can rephrase lemmas 6.2 and 6.3 in a simpler way, respectively:

- h is a forward morphism between coalgebras α and γ iff

$$h \cdot \xleftarrow{\gamma} \subseteq \xleftarrow{\alpha} \cdot h \tag{6.9}$$

- h is a backward morphism between coalgebras α and γ iff

$$\xleftarrow{\gamma} \cdot h \subseteq h \cdot \xleftarrow{\alpha} \tag{6.10}$$

This makes it posible to carry out coalgebraic refinement via relational methods. The section which follows is concerned with this approach.

6.4 A Single Complete Rule for Data Refinement

The title of this section is intentionally that of a paper [58] which proves (using predicate transformers) that a single complete method is enough, instead of the two standard methods of data (relational) refinement.

Below we shall infer that for proving coalgebraic refinement we only need a single complete data refinement method, ie. downward simulation, this time using relations.

Lemma 6.4 (Downward Simulation) *For proving coalgebraic refinement we only need downward simulation, since we can reduce upward simulation given by the proof rule*

$$h \cdot \xleftarrow{\gamma} \subseteq \xleftarrow{\alpha} \cdot h \tag{6.11}$$

to

$$\xleftarrow{\gamma} \cdot h^\circ \subseteq h^\circ \cdot \xleftarrow{\alpha} \tag{6.12}$$

ie. downward simulation for a relation which is the converse of a function.
Proof:

$$
\begin{array}{cl}
 & h \textit{ is an upward simulation} \\
\equiv & \qquad \{ \textit{ by definition of upward simulation} \} \\
 & h \cdot \xleftarrow{\gamma} \subseteq \xleftarrow{\alpha} \cdot h \\
\equiv & \qquad \{ \textit{ shunting 2 times} \} \\
 & \xleftarrow{\gamma} \cdot h^\circ \subseteq h^\circ \cdot \xleftarrow{\alpha} \\
\equiv & \qquad \{ \textit{ by definition of downward simulation} \} \\
 & h^\circ \textit{ is a downward simulation}
\end{array}
$$

6.4.1 Coalgebraic Refinement Via Single Complete Rule by Calculation

We will do coalgebraic refinement by calculation according to the strategy which follows. Aiming at refining a given coalgebra α via its transition relation $\xleftarrow{\alpha}$, we use the calculus associated to the single complete rule (downward simulation, Lemma 6.4) and calculate a refinement $\xleftarrow{\gamma}$ from $\xleftarrow{\alpha}$. In this way we obtain a refinement of the initial coalgebra, ie. γ.

Introduction to the Calculus. We shall present a calculus related with the single complete rule, ie. downward simulation, which we sketch below, which we express by a preorder as in the SETS calculus. Laws of this calculus are inequations of the form

$$\gamma \preceq_R \alpha$$

meaning (6.10) or (6.12), so $R = h$ or $R = h^\circ$.

$\preceq$ is a pre-order. The relation $\preceq$ is reflexive,

$$\alpha \preceq_{id} \alpha$$

and transitive,

$$\alpha \preceq_R \beta \wedge \beta \preceq_S \gamma \Rightarrow \alpha \preceq_{R \cdot S} \gamma$$

cf. the previous proof that bisimulation is closed under relational composition, ie.

$$
\begin{array}{ll}
 & \alpha \preceq_{R \cdot S} \gamma \\
\equiv & \quad \{ \text{by definition of } \preceq \} \\
 & \xleftarrow{\alpha} \cdot R \cdot S \subseteq R \cdot S \cdot \xleftarrow{\gamma} \\
\Leftarrow & \quad \{ \beta \preceq_S \gamma \text{ and transitivity of inclusion} \} \\
 & \xleftarrow{\alpha} \cdot R \cdot S \subseteq R \cdot \xleftarrow{\beta} \cdot S \\
\Leftarrow & \quad \{ \text{monotonicity of } (\cdot S) \} \\
 & \xleftarrow{\alpha} \cdot R \subseteq R \cdot \xleftarrow{\beta} \\
\equiv & \quad \{ \alpha \preceq_R \beta \} \\
 & True
\end{array}
$$

The statement of transitivity is decomposed in the following two formulations, for

- functions

$$\alpha \preceq_h \beta \wedge \beta \preceq_g \gamma \Rightarrow \alpha \preceq_{h \cdot g} \gamma$$

- converses of functions

$$\alpha \preceq_{h^\circ} \beta \wedge \beta \preceq_{g^\circ} \gamma \Rightarrow \alpha \preceq_{(g \cdot h)^\circ} \gamma$$

Bisimilarity. Let us relate downward simulation with bisimilarity, the union of all bisimulations.

Lemma 6.5 (Relation with bisimilarity) *The $\preceq$ preorder is related with bisimilarity as follows:*

$$\alpha \preceq_{f^\circ} \alpha \;\wedge\; \alpha \preceq_f \alpha \quad \equiv \quad \alpha \sim_f \alpha$$

Proof:

$$
\begin{array}{ll}
& \alpha \preceq_{f^\circ} \alpha \wedge \alpha \preceq_f \alpha \\
\equiv & \qquad \{ \textit{by definition of } \preceq \} \\
& \xrightarrow{\alpha} \cdot f^\circ \subseteq f^\circ \cdot \xrightarrow{\alpha} \wedge \xrightarrow{\alpha} \cdot f \subseteq f \cdot \xrightarrow{\alpha} \\
\equiv & \qquad \{ \textit{shunting} \} \\
& f \cdot \xrightarrow{\alpha} \subseteq \xrightarrow{\alpha} \cdot f \wedge \xrightarrow{\alpha} \cdot f \subseteq f \cdot \xrightarrow{\alpha} \\
\equiv & \qquad \{ \textit{ping-pong} \} \\
& f \cdot \xrightarrow{\alpha} = \xrightarrow{\alpha} \cdot f \\
\equiv & \qquad \{ \textit{homomorphism on } \mathsf{F}\textit{-coalgebra } \alpha\textit{, (6.8)} \} \\
& \mathsf{F}\, f \cdot \alpha = \alpha \cdot f \\
\equiv & \qquad \{ \textit{(endo)homomorphism iff (functional) bisimulation, see Lemma 6.1} \} \\
& \alpha \sim_f \alpha
\end{array}
$$

Since for the general case a homomorphism is a functional bisimulation [132], we can state the following corollary:

Corollary 6.1

$$\alpha \preceq_{f^\circ} \gamma \ \wedge \ \gamma \preceq_f \alpha \quad \equiv \quad \alpha \sim_f \gamma$$

The Calculus is structural. We begin by formulating the monotonicity laws of the main relational operators. For this we need to make transition relations explicit. So, instead of a preorder on coalgebras we refer to the corresponding preorder on transition relations,

$$\xleftarrow{\gamma} \sqsubseteq_R \xleftarrow{\alpha}$$

meaning (6.10) or (6.12), so $R = h$ or $R = h^\circ$.

Lemma 6.6 (Monotonicity of converse) *Converse is monotonic:*

$$\xleftarrow{\gamma} \sqsubseteq_{h^\circ} \xleftarrow{\alpha} \quad \equiv \quad \xleftarrow{\gamma}^\circ \sqsubseteq_{h^\circ} \xleftarrow{\alpha}^\circ$$

Proof:

$$
\begin{array}{ll}
 & \xleftarrow{\gamma} \sqsubseteq_{h^\circ} \xleftarrow{\alpha} \\
\equiv & \quad \{\textit{ by definition of } \sqsubseteq\} \\
 & \xleftarrow{\gamma} \cdot h^\circ \subseteq h^\circ \cdot \xleftarrow{\alpha} \\
\equiv & \quad \{\textit{ converse}\} \\
 & h \cdot \xleftarrow{\gamma}^\circ \subseteq \xleftarrow{\alpha}^\circ \cdot h \\
\equiv & \quad \{\textit{ shunting}\} \\
 & \xleftarrow{\gamma}^\circ \cdot h^\circ \subseteq h^\circ \cdot \xleftarrow{\alpha}^\circ \\
\equiv & \quad \{\textit{ by definition of } \sqsubseteq\} \\
 & \xleftarrow{\gamma}^\circ \sqsubseteq_{h^\circ} \xleftarrow{\alpha}^\circ
\end{array}
$$

Note that monotonicity of converse cannot be proved when we replace h° by h.

Meet and Join. We can prove some other results relative to join and meet.

Lemma 6.7 (Monotonicity of $\cup$) *If*

$$\xleftarrow{\gamma} \sqsubseteq_R \xleftarrow{\alpha}$$

and

$$\xleftarrow{\beta} \sqsubseteq_R \xleftarrow{\delta}$$

then

$$(\xleftarrow{\gamma} \cup \xleftarrow{\beta}) \sqsubseteq_R (\xleftarrow{\alpha} \cup \xleftarrow{\delta})$$

Proof:

$$
\begin{array}{ll}
 & \xleftarrow{\gamma} \sqsubseteq_R \xleftarrow{\alpha} \wedge \xleftarrow{\beta} \sqsubseteq_R \xleftarrow{\delta} \\
\equiv & \quad \{\textit{ definition of } \sqsubseteq\} \\
 & \xleftarrow{\gamma} \cdot R \subseteq R \cdot \xleftarrow{\alpha} \wedge \xleftarrow{\beta} \cdot R \subseteq R \cdot \xleftarrow{\delta} \\
\Rightarrow & \quad \{\textit{ monotonicity of } \cup\} \\
 & \xleftarrow{\gamma} \cdot R \cup \xleftarrow{\beta} \cdot R \subseteq R \cdot \xleftarrow{\alpha} \cup R \cdot \xleftarrow{\delta} \\
\equiv & \quad \{\ (R\cdot) \textit{ and } (\cdot R) \textit{ are lower adjoints}\} \\
 & (\xleftarrow{\gamma} \cup \xleftarrow{\beta}) \cdot R \subseteq R \cdot (\xleftarrow{\alpha} \cup \xleftarrow{\delta}) \\
\equiv & \quad \{\textit{ definition of } \sqsubseteq\} \\
 & \xleftarrow{\gamma} \cup \xleftarrow{\beta} \sqsubseteq_R \xleftarrow{\alpha} \cup \xleftarrow{\delta}
\end{array}
$$

Lemma 6.8 (Monotonicity of $\cap$) *If*

$$\xleftarrow{\gamma} \sqsubseteq_R \xleftarrow{\alpha}$$

and

$$\xleftarrow{\beta} \sqsubseteq_R \xleftarrow{\delta}$$

then

$$(\xleftarrow{\gamma} \cap \xleftarrow{\beta}) \sqsubseteq_R (\xleftarrow{\alpha} \cap \xleftarrow{\delta})$$

if the following conditions hold,

$$(\ker R)\cdot \xleftarrow{\alpha} \subseteq \xleftarrow{\alpha} \vee (\ker R)\cdot \xleftarrow{\delta} \subseteq \xleftarrow{\delta}$$

and

$$\xleftarrow{\gamma} \cdot img\, R \subseteq \xleftarrow{\gamma} \vee \xleftarrow{\beta} \cdot img\, R \subseteq \xleftarrow{\beta}$$

Proof:

$$
\begin{array}{ll}
 & \xleftarrow{\gamma} \sqsubseteq_R \xleftarrow{\alpha} \wedge \xleftarrow{\beta} \sqsubseteq_R \xleftarrow{\delta} \\
\equiv & \quad \{ \textit{ definition of } \sqsubseteq \} \\
 & \xleftarrow{\gamma} \cdot R \subseteq R\cdot \xleftarrow{\alpha} \wedge \xleftarrow{\beta} \cdot R \subseteq R\cdot \xleftarrow{\delta} \\
\Rightarrow & \quad \{ \textit{ monotonicity of } \cap \} \\
 & \xleftarrow{\gamma} \cdot R \cap \xleftarrow{\beta} \cdot R \subseteq R\cdot \xleftarrow{\alpha} \cap R\cdot \xleftarrow{\delta} \\
\equiv & \quad \{ \textit{ conditional laws (B.15) and (B.16} \} \\
 & (\xleftarrow{\gamma} \cap \xleftarrow{\beta}) \cdot R \subseteq R \cdot (\xleftarrow{\alpha} \cap \xleftarrow{\delta}) \\
\equiv & \quad \{ \textit{ definition of } \sqsubseteq \} \\
 & \xleftarrow{\gamma} \cap \xleftarrow{\beta} \sqsubseteq_R \xleftarrow{\alpha} \cap \xleftarrow{\delta}
\end{array}
$$

Lemma 6.8 is a little complicated. We propose the following alternative:

Lemma 6.9 (Monotonicity of $\cap$) *If*

$$\xleftarrow{\alpha} \cdot f \subseteq f^{\circ} \cdot \xleftarrow{\beta}$$

and

$$\xleftarrow{\gamma} \cdot f \subseteq f^{\circ} \cdot \xleftarrow{\delta}$$

then

$$(\xleftarrow{\alpha} \cap \xleftarrow{\gamma}) \cdot f \subseteq f^{\circ} \cdot (\xleftarrow{\beta} \cap \xleftarrow{\delta})$$

Proof:

$$
\begin{array}{ll}
& \xleftarrow{\alpha} \cdot f \subseteq f^\circ \cdot \xleftarrow{\beta} \wedge \xleftarrow{\gamma} \cdot f \subseteq f^\circ \cdot \xleftarrow{\delta} \\
\equiv & \quad \{ \textit{ monotonicity of} \cap \} \\
& \xleftarrow{\alpha} \cdot f \cap \xleftarrow{\gamma} \cdot f \subseteq f^\circ \cdot \xleftarrow{\beta} \cap f^\circ \cdot \xleftarrow{\delta} \\
\equiv & \quad \{ \ (f^\circ \cdot) \textit{ and } (\cdot f) \textit{ are upper adjoints} \} \\
& (\xleftarrow{\alpha} \cap \xleftarrow{\gamma}) \cdot f \subseteq f^\circ \cdot (\xleftarrow{\beta} \cap \xleftarrow{\delta})
\end{array}
$$

If $f = f^\circ$ we have the monotonicity result,
if

$$\xleftarrow{\alpha} \sqsubseteq_f \xleftarrow{\beta}$$

and

$$\xleftarrow{\gamma} \sqsubseteq_f \xleftarrow{\delta}$$

then

$$(\xleftarrow{\alpha} \cap \xleftarrow{\gamma}) \sqsubseteq_f (\xleftarrow{\beta} \cap \xleftarrow{\delta})$$

Composition. Now considering another relational operator, composition, we prove its monotonicity relative to downward simulation.

Lemma 6.10 (Monotonicity of composition) *If*

$$\xleftarrow{\gamma} \sqsubseteq_R \xleftarrow{\alpha}$$

and

$$\xleftarrow{\beta} \sqsubseteq_R \xleftarrow{\delta}$$

then

$$(\xleftarrow{\gamma} \cdot \xleftarrow{\beta}) \sqsubseteq_R (\xleftarrow{\alpha} \cdot \xleftarrow{\delta})$$

Proof:

$$
\begin{array}{ll}
& \xleftarrow{\gamma} \cdot \xleftarrow{\beta} \cdot R \\
\subseteq & \quad \{ \textit{ by hypothesis} \} \\
& \xleftarrow{\gamma} \cdot R \cdot \xleftarrow{\delta} \\
\subseteq & \quad \{ \textit{ by hypothesis} \} \\
& R \cdot \xleftarrow{\alpha} \cdot \xleftarrow{\delta}
\end{array}
$$

Relators. Finally, a property involving an arbitrary relator F.

Lemma 6.11 (F-monotonicity) *If*

$$\xleftarrow{\gamma} \sqsubseteq_R \xleftarrow{\alpha}$$

then

$$\mathsf{F} \xleftarrow{\gamma} \sqsubseteq_{\mathsf{F}R} \mathsf{F} \xleftarrow{\alpha}$$

Proof:

$$\begin{array}{cl} & \xleftarrow{\gamma} \sqsubseteq_R \xleftarrow{\alpha} \\ \equiv & \qquad \{ \textit{ by definition of } \sqsubseteq \} \\ & \xleftarrow{\gamma} \cdot R \subseteq R \cdot \xleftarrow{\alpha} \\ \Rightarrow & \qquad \{ \mathsf{F}\textit{-monotonicity} \} \\ & \mathsf{F}(\xleftarrow{\gamma} \cdot R) \subseteq \mathsf{F}(R \cdot \xleftarrow{\alpha}) \\ \Rightarrow & \qquad \{ \mathsf{F} \textit{ commutes with composition} \} \\ & (\mathsf{F} \xleftarrow{\gamma}) \cdot \mathsf{F}R \subseteq \mathsf{F}R \cdot \mathsf{F} \xleftarrow{\alpha} \\ \equiv & \qquad \{ \textit{ by definition of } \sqsubseteq \} \\ & \mathsf{F} \xleftarrow{\gamma} \sqsubseteq_{\mathsf{F}R} \mathsf{F} \xleftarrow{\alpha} \end{array}$$

Corollary 6.2 *If*

$$\xleftarrow{\gamma} \sqsubseteq_R \xleftarrow{\alpha}$$

and

$$\xleftarrow{\beta} \sqsubseteq_S \xleftarrow{\delta}$$

then

$$(\xleftarrow{\gamma} \times \xleftarrow{\beta}) \sqsubseteq_{R \times S} (\xleftarrow{\alpha} \times \xleftarrow{\delta})$$

Corollary 6.3 *If*

$$\xleftarrow{\gamma} \sqsubseteq_R \xleftarrow{\alpha}$$

and

$$\xleftarrow{\beta} \sqsubseteq_S \xleftarrow{\delta}$$

then

$$(\xleftarrow{\gamma} + \xleftarrow{\beta}) \sqsubseteq_{R+S} (\xleftarrow{\alpha} + \xleftarrow{\delta})$$

Summary. As we have seen, in the calculus of simulations presented above we don't need to define complicated downward simulations. We just need to know that id is a downward simulation and order the transition relations by (converse of) inclusion.

Let's see what this means. We know that the converse of inclusion is algorithmic refinement for entire relations, cf. (5.29). We know also that it is reasonable to assume that we are dealing with entire relations, the totalized (transposed) versions of every relation. So we have found a simple method of refinement over transition relations which justifies the option of doing coalgebraic refinement indirectly by downward simulation of transition relations.

6.5 Isomorphism between Coalgebras and Transition Relations

F-transition relations are defined as in (6.7). There is an isomorphism between F-coalgebras and F-transition relations, which comes after transposition for the case of F-coalgebras and F-transition relations.

Lemma 6.12 (Transposition for F-coalgebras and F-transition Relations)

$$\alpha = \Gamma_{\mathsf{F}} \xrightarrow{\alpha} \quad \equiv \quad \xrightarrow{\alpha} = \epsilon_{\mathsf{F}} \cdot \alpha \tag{6.13}$$

Proof: *In the context of transposition, as given by the following universal property*

$$f = \Gamma_{\mathsf{F}}\ R \equiv \epsilon_{\mathsf{F}} \cdot f = R$$

we can prove:

$$\alpha = \Gamma_{\mathsf{F}} \xrightarrow{\alpha} \tag{6.14}$$

as shown in the following reasoning,

$$\begin{array}{ll}
 & \Gamma_{\mathsf{F}} \xrightarrow{\alpha} \\
= & \qquad \{\ \textit{transition relation}\} \\
 & \Gamma_{\mathsf{F}}(\epsilon_{\mathsf{F}} \cdot \alpha) \\
\equiv & \qquad \{\ \textit{fusion (3.11), since } \Gamma_{\mathsf{F}}\epsilon_{\mathsf{F}} \cdot \alpha \textit{ is a function }\} \\
 & \Gamma_{\mathsf{F}}\epsilon_{\mathsf{F}} \cdot \alpha \\
= & \qquad \{\ \textit{reflection (3.10)}\} \\
 & id \cdot \alpha \\
= & \qquad \{\ \textit{trivial}\} \\
 & \alpha
\end{array}$$

From (6.7) and (6.14) we are led to (6.13).

Corollary 6.4 (Isomorphism)

$$\alpha \quad = \quad \Gamma_{\mathsf{F}} \xrightarrow{\alpha} \tag{6.15}$$

6.6 Universal Coalgebra and Refinement Concepts by Example

We finish this chapter giving two examples of refinement in the context of coalgebras.

6.6.1 Coalgebraic Refinement

Let us consider the following example of a transition system S, taken from [132], such that its power transpose is a $\mathcal{P}(A \times id)$ coalgebra α, which we want to implement.

$$
\begin{tikzcd}
s_0 \arrow[r, "b"] \arrow[d, "a"] & s_1 \arrow[r, "b"] \arrow[d, "a"] & \cdots \\
s_0' & s_1' &
\end{tikzcd}
$$

To this coalgebra we associate transition relation $\xrightarrow{\alpha}$, such that

$$\xrightarrow{\alpha} = \epsilon_{\mathcal{P}(A \times id)} \cdot \alpha$$

The membership $\epsilon_{\mathcal{P}(A \times id)}$ is calculated in the following way:

$$
\begin{array}{ll}
 & \epsilon_{\mathcal{P}(A \times id)} \\
= & \quad \{ \text{ composition (2.30)} \} \\
 & \epsilon_{\mathcal{P}} \cdot \epsilon_{A \times id} \\
= & \quad \{ \text{ powerset and product (2.28)} \} \\
 & \epsilon \cdot (\epsilon_A \cdot \pi_1 \cup \epsilon_{id} \cdot \pi_2) \\
= & \quad \{ \text{ constant and identity functors (2.26, 2.27)} \} \\
 & \epsilon \cdot (\perp \cdot \pi_1 \cup id \cdot \pi_2) \\
= & \quad \{ \text{ trivial} \} \\
 & \epsilon \cdot \pi_2
\end{array}
$$

So

$$\xrightarrow{\alpha} = \epsilon \cdot \pi_2 \cdot \alpha$$

We want to calculate an implementation $\xrightarrow{\gamma}$ such that:

$$\xrightarrow{\gamma} \subseteq \xrightarrow{\alpha} \tag{6.16}$$

corresponding to downward simulation id:

$$\xrightarrow{\gamma} \cdot id \quad \subseteq \quad id \cdot \xrightarrow{\alpha} \tag{6.17}$$

that is, the following inequation,

$$\gamma \preceq_{id} \alpha \tag{6.18}$$

We assume that transition relations are entire (the obvious totalization is assumed), so the implementation is done by reduction of non-determinism.

An obvious implementation at the level of transition relations is obtained as follows,

$$\xrightarrow{\gamma} = \xrightarrow{\alpha} - \{(s_i, s'_i) | i \geq 0\}$$

To the transition relation $\xrightarrow{\gamma}$ we associate the obvious LTS R, a completely deterministic one, depicted as follows:

$$\begin{tikzcd} s_0 \arrow[r, "b"] & s_1 \arrow[r, "b"] & \cdots \\ s'_0 & s'_1 & \end{tikzcd}$$

Implementation coalgebra γ, the power transpose of LTS R, is such that

$$(\Gamma_{\mathsf{T}} \xrightarrow{\gamma}) \cdot id \mathrel{\dot{\leq}} \mathsf{T}\, id \cdot (\Gamma_{\mathsf{T}} \xrightarrow{\alpha}) \tag{6.19}$$

according to (6.17) and Lemma 6.3 and $\mathsf{T} = \mathcal{P}(A \times Id)$.

Inequation (6.19) simplifies to:

$$\gamma \mathrel{\dot{\leq}} \alpha \tag{6.20}$$

since

$$\mathsf{T}\, id = id$$

as T is a functor and preserves identity, and 6.14.

6.6.2 From Data Refinement in Full to Coalgebraic Refinement

As another example, we resume work on the *Buffer* example of the previous chapter. Recall (5.44) which expresses the full data refinement of operation $BufferIn$ by $arrayIn$ as required. By (5.40) this is equivalent to the following upward simulation

$$(rb \times id) \cdot arrrayIn \subseteq BufferIn \cdot (rb \times id)$$

which can be reduced to the single complete rule (ie. downward simulation)

$$arrrayIn \cdot (rb \times id)^{\circ} \subseteq (rb \times id)^{\circ} \cdot BufferIn$$

that is

$$arrrayIn \quad \sqsubseteq_{(rb \times id)^{\circ}} \quad BufferIn$$

By Lemma 6.2 this is equivalent to forward morphism:

$$(\Gamma_{\mathsf{T}}\ arrrayIn) \cdot (rb \times id) \mathrel{\dot{\leq}} T(rb \times id) \cdot (\Gamma_{\mathsf{T}}\ BufferIn)$$

where operation $arrayIn$ is taken as a non-deterministic transition relation and $BufferIn$ is a completely deterministic transition relation, as we consider it now.

Of course, T-coalgebra

$$(\Gamma_{\mathsf{T}}\ BufferIn)$$

is implemented by T-coalgebra

$$(\Gamma_{\mathsf{T}}\ arrayIn)$$

6.7 Summary

We proved that coalgebraic refinement, as witnessed by a forward morphism or a backward morphism, is united (equivalent) to relational (data) refinement [32], as witnessed by an upward simulation or a downward simulation, respectively. We also proved that we only need a single complete rule for data refinement (to prove coalgebraic refinement). This is because an upward simulation h is equivalent to a downward simulation h°. We defined the preorder $\preceq$ that relates coalgebras and which is related with bisimilarity of coalgebras. We proved that related relation $\sqsubseteq$ admits structural reasoning and is monotonic relative to the main relational operators. In this calculus of simulations, we don't need to define complicated downward simulations. We just need to know that id is a downward simulation and order the transition relations by inclusion.

Chapter 7

Conclusions and Future Work

This dissertation addresses the foundations of program development by *calculation*. Its background is the binary pointfree (PF) relational algebra.

The following section, written in the style of an epilogue, summarizes the work developed in the previous chapters. This is followed by two other sections, one emphasizing the main contributions and the other presenting directions for future work.

7.1 Epilogue

Relational algebra appears as a generic universe of discourse able to provide a foundation for computing science. Within relational algebra, it is possible to work with predicates and sets as binary relations. Relational algebra incorporates logic via the PF-transform, which translates first order logic formulas to point-free binary relation expressions. Unary predicates are encoded as coreflexive relations. In this way, binary relation algebra also incorporates set theory, since the meaning of a set is the meaning of its *characteristic* predicate. So, the subsets of a given set A are in one-one correspondence with the coreflexives at most id_A.

Tarski [137] is among those who gave new energy to the calculus of relations inherited from the nineteenth century in his development of "set theory without variables". This advice was followed eventually by the algebra of programming school [29, 12]. The priority of this discipline has been, however, mostly on reasoning about *algorithms* rather than *data structures*. Attempts to set up a *calculus of data structures* date back to [104, 105, 106]. The approach, however, was not agile enough. And the issue of program refinement as such is not fully covered.

This dissertation addresses program refinement from a unified perspective: that which expresses and reasons about programming ingredients using binary relations only. As these include functions as particular cases, the universe of functional programming is implicitly covered. In other words, relations are not only viewed as abstract devices expressing general phenomena, which include transposition, membership and monads, but also used to specify both data structures and (eg. functional) algorithms present in computer programs.

7.2 Summary of Contributions

The dissertation is structured in two main parts: mathematical background and refinement calculi. The main contribution in the first part is the proposal of a generic transpose construction which serves two main purposes:

- It makes it possible to characterize particular classes of relations as monadic functions and thus takes advantage of the algebra of such functions in reasoning about relations within each such class. Transposition is, in a sense, employed in this dissertation as a *leit-motiv*.

- It bridges with the coalgebraic theory of systems, making it possible to study coalgebraic refinement relationally in the second part of the dissertation.

The second part's main contributions are the following:

- Specification of data structures as (finite) binary relations which are the subject of data refinement rules.

- Development of a data refinement calculus based on connected abstraction/ representation pairs, based on a refinement preorder denoted by $\leq$.

- Identification of an important subset of such rules which are perfect Galois connections, and proposal of a method for calculating the concrete invariants induced by such rules.

- Development of a pointfree relational approach to Groves' factorization of the algorithmic refinement ordering ($\vdash$), in a way which makes it possible to take into account the meet of two opposites: *increase of definition* versus *decrease of non-determinism*.

- Development of a single complete rule for coalgebraic algorithmic refinement which unifies previous attempts to calculating transition relations associated to coalgebras.

All in all, this dissertation binds together two main refinement concepts and theories found in the literature by encoding them in the form of two preorders:

- the preorder $\leq$ of refinement of data structures and

- the preorder $\vdash$ of algorithmic refinement.

The outcome is a uniform treatment of such theories eased up by the calculational style brought about by the pointfree transform. This includes the full data refinement concept which is equivalent to standard data refinement, to downward simulation and to upward simulation. In turn, coalgebraic refinement (in its two variants, forward morphisms and backward morphisms) is shown to be equivalent to standard data refinement, upward simulation and downward simulation, respectively.

In summary, *downward simulation* is the refinement concept which unifies algebraic and coalgebraic implementation, and sequential and concurrent computation.

7.3 Future Work

This section presents prospects for future work brought about by the main research directions presented in this dissertation.

7.3.1 Transposition

The theory of relational transposition presented in chapter 3 can be generalized via the categorial notion of an adjunction [23]. For instance, there is an adjunction between the categories Set and Rel, as depicted in the following diagram:

$$\frac{A \longrightarrow E\ B}{J\ A \longrightarrow B}$$

where J is the inclusion of Set in Rel and E is the existential image functor [29].

The power transpose can be defined in the context of this adjunction. The Maybe transpose can be defined in the context of a new adjunction, this time involving the categories Set and Par (the category of partial functions or simple relations).

Since an adjunction always induces a monad, we should frame the generic monad definition given by (3.38, 3.39) in section 3.6 into the generic theory.

7.3.2 Data Refinement Laws as Galois Connections

Normalization "is" Data Refinement. In the past we wrote a technical report [128] which presents a constructive approach to relational database normalization theory [84, 114]. It presents a set of SETS laws which prevent from the violation of normal forms caused by partial dependencies (2NF), transitive dependencies (3NF) and multivalued dependencies (4NF). They eliminate a "semantic" datatype invariant in the refinement process corresponding to the normalization task taking place.

A future development will be to rework such an approach in the context of the refinement laws studied in chapter 4 of this dissertation. For instance, in the context of [129] we found the need for refinement inequations involving invariants, eg. ϕ in:

$$\begin{tikzcd} A_\phi \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "F"] \end{tikzcd} \quad \equiv \quad \phi \subseteq F \cdot R \subseteq id$$

where ϕ should be understood as a coreflexive relation. This will open a new line of research where coreflexive relations "replace " datatypes with invariants in a way which makes calculations easier to carry out.

Abstract Interpretation. Abstract interpretation is based on the use of Galois connections, as introduced by P. Cousot and R. Cousot [41, 42], to establish the correspondence between domains of concrete or exact properties and domains of abstract or approximate properties [43]. K. Backhouse and R. Backhouse [10] have addressed this theme relationally via extensive use of Galois connections to guarantee safety of abstract interpretations.

It will be interesting to study the collection of perfect Galois connections presented in section 4.5 once framed in the abstract interpretation setting.

7.3.3 Algorithmic Refinement

Other Approaches to Algorithmic Refinement. One should pay attention to other approaches to algorithmic refinement, in particular to that of M. Fokkinga [56] concerning simulations and compositional refinement and involving the notion of a *context*, ie. a program with a hole in it where other program parts can be plugged in.

Another approach is that of T. Nipkow [102] who develops two theories of data abstraction and refinement, one for applicative types as found in functional languages and the other for state based types as found in imperative programming languages.

Finally, we should mention a paper by W. Kahl [80] which attempts to give a general approach to relational specification and refinement concepts, and presents a selection of program derivation calculi based on these. One which is particularly interesting is *demonic* data refinement.

Rely-guarantee conditions. Rely-guarantee conditions are part of a method which includes specifying concurrent computations on shared memory, by controlling the *interference* among them. This method also addresses the *compositional* development of parallel imperative programs which implement the rely-guarantee specifications of it. In [130] we have done some preliminary work on applying pointfree binary relational algebra to prove (by calculation) the soundness of rely-guarantee logic associated to this method. We had in mind the alternative proof of [40] (see also [39]) which motivated the development of a less complicated one. The proof in [71] (see also [49]) is shorter than [40]'s but it's not calculational like the one we have done. The proof in [101] (see also [100]) was carried out with the support of a theorem prover (Isabelle/HOL), but without the algebraic nature of pointfree binary relational calculus.

7.3.4 Coalgebraic Refinement

Lemma 6.12 in this dissertation generalizes the universal property upon which [126] lift weak bisimilarity from the relation to the coalgebraic level. This offers a wider perspective on approaching universal coalgebra relationally and calls for further research. This should be carried out in deeper involvement with universal coalgebra theory [132, 4], namely in what concerns subsystems, coinductive definitions (cf. anamorphisms) and coinductive proofs [19, 20].

For coalgebraic refinement reference [21] is a permanent source of inspiration. The work of Jacobs in this context [74] should also be taken into account, in particular in the definition of a coalgebraic specification language [73]. The integration of algebraic and coalgebraic methods in specification is also addressed in [38].

Pointfree Factorization of Coalgebraic Refinement. The work reported in chapter 5 should be regarded as a step towards a broader research aim: that of developing a clear-cut PF-theory of *coalgebraic* refinement. The intuition is provided by formula

(5.1) once again, whose set-valued functions can be regarded *both* as power-transposes of binary relations [117] and coalgebras of the powerset functor. Instead of favouring the former view as in this dissertation , we want to exploit the latter and follow the approach of [87], who study refinement of software components modelled by coalgebras of functor $\mathsf{F}X = (O \times (\mathsf{B}X))^I$, where I and O model inputs and outputs and B is a monad describing the component's behaviour pattern.

The approach has already been treated generically in the pointfree style in [21], whereby set inclusion in (5.1) is generalized to a sub-preorder of F-membership-based inclusion. There is, however, no coalgebraic counterpart to the $\vdash_{pre}/\vdash_{post}$ factorization studied in the current dissertation. Such a generalization is a prompt topic for future research.

7.3.5 Additional features of Relational Algebra

Linear logic [59, 60] is a logic that complements classical logic and intuitionistic logic and which has some connections with concurrency, see eg. [37] which also relates state-based concurrency as found in Petri Nets [123, 28] and process-based concurrency as found in π-calculus [89, 133], through linear logic.

Barr [22] gives the Rel category as a model of linear logic. In particular, an encoding of π-calculus into linear logic is given by [37]. The tensor product and linear implication are both modeled in Rel by the Cartesian product (which is not the product of Rel). Going further in this direction would enable one to encode the π-calculus into Rel, bearing the promise of developing a calculational approach to the π-calculus, possibly including refinement.

Bibliography

[1] Chritiene Aarts, Roland Backhouse, Paul Hoogendijk, Ed Voermans, and Jaap van der Woude. A relational theory of datatypes, December 1992.

[2] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[3] Peter Aczel and Nax Mendler. A final coalgebra theorem. In D.H. Pitt, editor, *Category Theory and Computer Science*, Lecture Notes in Computer Science, pages 357–365. Springer-Verlag, 1989.

[4] J. Adamek. Introduction to coalgebra. *Theory and Applications of Categories*, 14:157–199, 2005.

[5] ATX. Algoritmo de colocação de professores, 2004?

[6] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[7] R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. New York: Springer-Verlag, 1998. Graduate Texts in Computer Science.

[8] R.J.R. Back. Correcteness preserving program refinements: Proof theory and applications. *Mathematical Center Tracts*, 131, 1980.

[9] R.J.R. Back and J. von Wright. Duality in specification languages: A lattice-theoretical approach. *Acta Informatica*, 27:583–625, 1990.

[10] K. Backhouse and R.C. Backhouse. Safety of abstract interpretations for free,via logical relations and Galois connections. *Science of Computer Programming*, 15(1–2):153–196, 2004.

[11] R.C. Backhouse. An exploration of the Bird-Meertens Formalism, July 1988. Department of Computing Science, Groningen University, The Netherlands.

[12] R.C. Backhouse. Mathematics of program construction, June 2004. With help and contributions by Marcel Bijsterveld and Henk Doornbos and Rik van Geldrop and Diethard Michaelis and Jaap van der Woude.

[13] R.C. Backhouse. Regular algebra applied to language problems, 2004. Available from `http://www.cs.nott.ac.uk/~rcb/papers/` (Extended version of *Fusion on Languages* published in ESOP 2001. Springer LNCS 2028, pp. 107–121.).

[14] R.C. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J. van der Woude. Polynomial relators. In *2nd Int. Conf. Algebraic Methodology and Software Technology (AMAST'91)*, pages 303–362. Springer LNCS, 1992.

[15] R.C. Backhouse and P.F. Hoogendijk. Final dialgebras: From categories to allegories. *Informatique Theorique et Applications*, 33(4/5):401–426, 1999. Presented at Workshop on Fixed Points in Computer Science, Brno, August 1998.

[16] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. CACM, 21(8):613–639, August 1978.

[17] R. Balzer, T. E. Cheatham, and C. Green. Software technology in the 1990's: Using a new paradigm. *Computer*, 15(16), November 1983.

[18] L. S. Barbosa. *Components as Coalgebras*. University of Minho, December 2001. Ph. D. thesis.

[19] L. S. Barbosa. Process calculi *à la* Bird-Meertens. In *CMCS'01 - Workshop on Coalgebraic Methods in Computer Science*, pages 47–66, Genova, April 2001. ENTCS, volume 44.4, Elsevier.

[20] L. S. Barbosa and J. N. Oliveira. Coinductive interpreters for process calculi. In *FLOPS 2002http://www.ipl.t.u-tokyo.ac.jp/FLOPS2002/ - 6th International Symposium on Functional and Logic Programming, University of Aizu, Aizu, Japan, September 15-17, 2002*, September 2002.

[21] L.S. Barbosa and J.N. Oliveira. Transposing partial components — an exercise on coalgebraic refinement. *Theoretical Computer Science*, 365(1):2–22, 2006.

[22] M. Barr. *-autonomous categories and linear logic. *Mathematical Structures in Computer Science*, 1:159–178, 1991.

[23] M. Barr and C. Wells. *Category Theory for Computing Science*. Les Publications Centre de Recherches Mathematiques, 1999. third edition.

[24] F. L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, and Pepper. P. *The Munich Project CIP. Volume II: The Program Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[25] F. L. Bauer and H. Wossner. *Algorithmic Language and Program Development*. Springer-Verlag, 1982.

[26] F.L. Bauer et al. *The Munich Project CIP Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[27] R. Berghammer and B. von Karger. Formal derivation of csp programs from temporal specifications. In *Mathematics of Program Construction*, volume 947 of *Lectures Notes in Computer Science*, pages 180–196. Springer-Verlag, 1995.

[28] E. Best and C. Fernandez C. *Nonsequential Processes: A Petri Net View*. Springer-Verlag, 1988.

[29] R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997. C. A. R. Hoare, series editor.

[30] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, NATO ASI Series Vol. F36. Springer-Verlag, 1987.

[31] R.S. Bird and L. Meertens. Two exercices found in a book on algorithmics. In L. Meertens, editor, *Program Specification and Transformation*, pages 451–458. North-Holland, 1987.

[32] Eerke Boiten and Willem-Paul de Roever. Getting to the Bottom of Relational Refinement: Relations and Correctness, Partial and Total. In R. Berghammer and B. Möller, editors, *7th International Seminar on Relational Methods in Computer Science (RelMiCS 7)*, pages 82–88. University of Kiel, May 2003.

[33] Eerke Boiten and John Derrick. Unifying concurrent and relational refinement. In John Derrick, Eerke Boiten, Jim Woodcock, and Joakim von Wright, editors, *REFINE 02-The BCS FACS Refinement Workshop*, volume 70(3). Elsevier Science Publishers, July 2002.

[34] Christie Bolton, Jim Davies, and Jim Woodcock. On the refinement and simulation of data types and processes. In *IFM*, pages 273–292. Springer, 1999.

[35] Noureddine Boudriga, Fathi Elloumi, and Ali Mili. On the lattice of specifications: Applications to a specification methodology. *Formal Asp. Comput.*, 4(6):544–571, 1992.

[36] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM* :, 24(1):44–67, January 1977.

[37] I. Cervesato and A Scedrov. Relating state based and process based concurrency through linear logic. *ENTCS*, 165:145–176, 2006.

[38] Corina Cirstea. *Integrating Observations and Computations in the Specification of State-Based Dynamical Systems*. Oxford University, 2000. D. Phil. thesis.

[39] J.W. Coleman and C.B. Jones. Garanteeing the soundness of rely/garantee rules. Technical Report Series CS-TR-955, Department of Computing Science, University of Newcastle upon Tyne, March 2006.

[40] J.W. Coleman and C.B. Jones. A structural proof of the soundness of rely/garantee rules. *Journal of Logic and Computation*, 17(4):807–841, 2007.

[41] P. Cousot and Cousot R. Abstract interpretation: A unified lattice model for static analisys of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252. January 1977.

[42] P. Cousot and Cousot R. Systematic design of program analisys frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282. January 1979.

[43] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 3(2):103–179, 1992.

[44] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[45] J. Darlington. Program transformation. In *Funct. Prog. and Its Applications: An Advanced Course*. Cambridge Univ. Press, 1982.

[46] J. Darlington. *The Design of Efficient Data Representations*, chapter 7, pages 139–156. Macmillan, London, 1984.

[47] A. De Morgan. On the syllogism no. iv and on the logic of relations. *Transactions of the Cambridge Philosophical Society*, 10:331–358, 1860. Reprinted in: [48].

[48] A. De Morgan. On the syllogism and other logical writings. Yale University Press, 1966.

[49] W.-P. de Roever, F. de Boer, U. Hanneman, J. Hoomman, Y.Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[50] W.-P. de Roever and K. Engelhardt. *Data Refinement Model-Oriented Proof methods and their Comparison*. Cambridge University Press, 1999. With the assistance of J. Coenen and K.-H. Buth and P. Gardiner and Y. Lakhnech and F. Stomp.

[51] Moshe Deutsche, Martin C. Henson, and Steve Reeves. Modular reasoning in z: scrutinizing monotonicity and refinement, 2006. Under consideration for publication in Formal Aspects of Computing.

[52] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[53] R. Eshuis and M.M. Fokkinga. Comparing refinements for failure and bissimulation semantics. *Fundamentica Informaticae*, 52(4):297–321, 2002.

[54] J. Fitzgerald and P.G. Larsen. *Modelling Systems: Practical Tools and Techniques for Software Development http://www.csr.ncl.ac.uk/modelling-book/*. Cambridge University Press, 1st edition, 1998.

[55] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19, pages 19–32. American Mathematical Society, 1967. Proc. Symposia in Applied Mathematics.

[56] Maarten Fokkinga. A summary of refinement theory. Technical report, September 1994.

[57] M.M. Fokkinga and L. Meertens. Adjunctions. Memoranda Informatica 94-31, University of Twente, June 1994.

[58] P.H.B. Gardiner and Carrol Morgan. A single complete rule for data refinement. *Formal Asp. Comput.*, 5(4):367–382, 1993.

[59] J.-Y. Girard. Linear logic. TCS , 50:1–102, 1987.

[60] J.-Y. Girard. Linear logic, its syntax and semantics. In Regnier Girard, Lafont, editor, *Advances in Linear Logic*, London Mathematical Society Lecture Notes Series 222. Cambridge University Press, 1995.

[61] Lindsay Groves. Refinement and the z schema calculus. *ENTCS*, 70(3), 2002. The BCS FACS Refinement Workshop, Denmark July 2002.

[62] Jifeng He. Process refinement. In *The Theory and Practice of Refinement*, pages 37–59. McDermid, 1989.

[63] Jifeng He and C. A. R. Hoare. Prespecification in data refinement. In *Data Refinement in a Categorical Setting*, volume PRG-90 of *Technical Monograph*. Oxford University Computing Laboratory, 1990.

[64] Jifengi He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *B. Robinet and R. Wilhelm, editors Proc. ESOP 86*, Lecture Notes in Computer Science, pages 187–196. Springer, 1986.

[65] C. A. R. Hoare. An axiomatic basis for computer programming. CACM, 12,10:576–580, 583, October 1969.

[66] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[67] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985. C. A. R. Hoare.

[68] C.A.R. Hoare, J.F. He, and J.W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25:71–76, 1987.

[69] Paul Hoogendijk. *A Generic Theory of Data Types*. PhD thesis, University of Eindhoven, The Netherlands, 1997.

[70] Paul F. Hoogendijk and Oege de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, 2000.

[71] J. Hooman, W. de Roever, P. Pandya, Q. Xu, P. Zhou, and H. Schepers. A compositional approach to concurrency and its applications, April 2003.

[72] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*. Computer Software Engineering Series. Pitman, 1978. E. Horowitz (Ed.).

[73] B. Jacobs J. Roth and H. Tews. The coalgebraic class specification language ccsl. *Journal of Universal Computer Science*, 7(2), 2001.

[74] Bart Jacobs and Hendrik Tews. Assertional and behavioural refinement in coalgebraic specification. *Electronic Notes in Theoretical Computer Science*, 47, 2001.

[75] C. B. Jones. *Software Development — A Rigorous Approach*. Series in Computer Science. Prentice-Hall International, 1980. C. A. R. Hoare.

[76] C. B. Jones. *Systematic Software Development Using VDM*. Series in Computer Science. Prentice-Hall International, 1986. C. A. R. Hoare.

[77] C. B. Jones. The search for tractable ways of reasoning about programs. Technical Report UMCS-92-4-4, Department of Computer Science, University of Manchester, March 1992.

[78] S.L. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003. Also published as a Special Issue of the Journal of Functional Programming, 13(1) Jan. 2003.

[79] I. S. Jourdan. Reificação de tipos abstractos de dados: Uma abordagem matemática. Master's thesis, University of Coimbra, 1992. (In Portuguese).

[80] Wolfram Kahl. Refinement and development of programs from relational specification. *ENTCS*, 44(3), 2003.

[81] E. Kreyszig. *Advanced Engineering Mathematics*. John Wiley & Sons, Inc., 6th edition, 1988.

[82] S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, New-York, 1971.

[83] R. D. Maddux. The origin of relation algebras in the develpment and axiomatization of the calculus of relations. *Studia Logica*, 50(3-4):421–455, 1991.

[84] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983. ISBN 0-914894-42-0.

[85] F. M. Martins and J. N. Oliveira. Archetype-oriented user interfaces. *Computer & Graphics*, 14(1):17–28, 1990.

[86] L. Meertens. Algorithmics – towards programming as a mathematical activity. In J.K. Lenstra J.W. de Bakker, M. Hazewinkel, editor, *CWI Symp. on Mathematics and Computer Science*, CWI Monographs Vol. 1, pages 289–334. North-Holland, 1986.

[87] Sun Meng and L.S. Barbosa. On refinement of generic state-based software components. In C. Rettray, S. Maharaj, and C. Shankland, editors, *10th Int. Conf. Algebraic Methods and Software Technology (AMAST)*, pages 506–520, Stirling, July 2004. Springer Lect. Notes Comp. Sci. 3116.

[88] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989. C. A. R. Hoare.

[89] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[90] B. Moeller. A survey of the project cip: Computer-aided, intuition guided programming wide spectrum language and program transformations. Technical Report TUM-I8406, T. Universität München, July 1984.

[91] B. Moeller. Relations as a program development language. In B. Möller, editor, *Constructing Programs from Specifications*, pages 373–397. North-Holland, 1991.

[92] B. Moeller. Derivation of graph and pointer algorithms. In S.A. Schuman, B. Möller, and H.A. Partsch, editors, *Formal Program Development*, number 755 in Lecture Notes in Computer Science, pages 123–160. Springer, 1993.

[93] C. Morgan. *Programming from Specification*. Series in Computer Science. Prentice-Hall International, 1990. C. A. R. Hoare, series editor.

[94] C. Morgan. *Programming from Specification*. Prentice-Hall International, 1998. Third Edition.

[95] C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.

[96] C. Morgan and T. Vickers. *On the Refinement Calculus*. FACIT. Springer-Verlag, 1994.

[97] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3), 1987.

[98] J. M. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.

[99] Shin-Cheng Mu and Richard Bird. Inverting functions as folds. In *MPC'02: Mathematics of Program Construction*, Lecture Notes in Computer Science. Springer, 2002.

[100] Leonor Prensa Nieto. *Verification of Parallel Programs with Owicki-Gries and Rely-Garantee Methods in Isabelle/HOL*. Institut fur Informatic der Technischen Universitaet Munchen, 2001. Ph. D. thesis.

[101] Leonor Prensa Nieto. The rely-garantee method in isabelle/hol. In *Proceedings of ESOP 2003*, volume 2618 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[102] T. Nipkow. Formal Verification of Data Type Refinement — Theory and Practice. In J.W. de Bakker, W.P. de Roever, and G. Rosenberg, editors, *Stepwise Refinement of Distributed Systems*, LNCS (430), pages 561–591. Springer-Verlag, 1990.

[103] J. N. Oliveira. The transformational paradigm as a means of smoothing abrupt softwar e design steps. Technical Report CCES-JNO:R2/85, U. Minho, December 1985.

[104] J. N. Oliveira. Refinamento transformacional de especificações (terminais). In *Actas das XII "Jornadas Luso-Espanholas de Matemática"*, volume II, pages 412–417, Braga, Portugal, 4-8 May 1987.

[105] J. N. Oliveira. A Reification Calculus for Model-Oriented Software Specification www.di.uminho.pt/ jno//html/fac90.cr.html. *Formal Aspects of Computing*, 2(1):1–23, April 1990.

[106] J. N. Oliveira. Software Reification using the SETS Calculus www.di.uminho.pt/ jno//html/bcs92.cr.html. In Tim Denvir, Cliff B. Jones, and Roger C. Shaw, editors, *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. ISBN 0387197524, Springer-Verlag, 8–10 January 1992. (Invited paper).

[107] J. N. Oliveira. Hash Tables — A Case Study in $\trianglelefteq$-calculationwww.di.uminho.pt/ jno//html/94-12-1-.html. Technical Report DI/INESC 94-12-1, INESC Group 2361, Braga, December 1994.

[108] J. N. Oliveira. 'Fractal' Types: an Attempt to Generalize Hash Table Calculationwww.di.uminho.pt/ jno//html/wgp98.abs.html. In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, June 1998.

[109] J. N. Oliveira. First steps in pointfree functional dependency theory. Technical report, Departamento de Informática, Universidade do Minho, 2005.

[110] J.N. Oliveira. *Métodos Formais de Programação*. Departamento de Informática-Universidade do Minho, 1998.

[111] J.N. Oliveira. An introduction to data refinement, 2003. Formal Methods II, 2002/03.

[112] J.N. Oliveira. Calculate databases with 'simplicity', September 2004. Presentation at the IFIP WG 2.1 #59 Meeting, Nottingham, UK.

[113] J.N. Oliveira. An introduction to algorithmic refinement, June 2004.

[114] J.N. Oliveira. Data dependency theory made generic– by calculation, 2006. Presentation at the IFIP WG 2.1 #62 Meeting, Namur, Belgium.

[115] J.N. Oliveira. Métodos formais de programação ii, 2006. Sumários da disciplina.

[116] J.N. Oliveira. Data Transformation by Calculation../ps/gttse07.pdf. In João Saraiva Ralf Lämmel and Joost Visser, editors, *GTTSE 2007 Proceedings*, pages 139–198, June 2007. International Summer School, July 2-7.

[117] J.N. Oliveira and C.J. Rodrigues. Transposing relations: from *Maybe* functions to hash tables. In D. Kozen, editor, *MPC'07 http://www.cs.cornell.edu/Projects/MPC2004 : Seventh International Conference on Mathematics of Program Construction, 12-14 July, 2004, Stirling, Scotland, UK (Organized in conjunction with AMAST'04)*, volume 3125 of *Lecture Notes in Computer Science*, pages 334–356. Springer, July 2004.

[118] J.N. Oliveira and C.J. Rodrigues. Pointfree factorization of operation refinement. In *FM'06: 14 International Symposium on Formal Methods, 21-27 August, 2006, McMaster University, Hamilton, Ontario, Canada*, volume 4085 of *Lecture Notes in Computer Science*, pages 236–251. Springer, August 2006.

[119] D. Park. Concurrency and automata on infinite sequences. LNCS , 104:334–356, 1981.

[120] H. A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.

[121] C. S. Peirce. Description of a notation for the logic of relatives, resulting from an amplification of the conceptions of boole's calculus of logic. *Memoirs of the American Academy of Sciences*, 9:317–378, 1870. Reprinted in: [122].

[122] C. S. Peirce. Collected papers. Harvard University Press, 1933.

[123] C.A. Petri. *Kommunikation mit Automaten*. Bonn: Institut fur Instrumentelle Mathematik Schriften des IIM Nr. 2, 1962.

[124] V. Pratt. Origins of the calculus of binary relations. In *Proc. of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 248–254, Santa Cruz, CA, 1992.

[125] J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing 83*, pages 513–523, 1983.

[126] P. Ribeiro, L.S. Barbosa, and S. Wang. An exercise on transition systems. *Electronic Notes on Theoretical Computer Science*, 207:89–106, 2007.

[127] C. J. Rodrigues. Reificação e cálculos de reificação. Technical report, University of Minho, April 1995.

[128] C. J. Rodrigues and J. N. Oliveira. Normalization is Data Reificationwww.di.uminho.pt/ jno//html/cjr96.abs.html. Technical Report UMDITR9702, University of Minho, Dec. 1997.

[129] C.J. Rodrigues. Data refinement by calculation, 2005. Presented at the DI/UM Ph.D. Student Symposium, Quinta da Torre, Soutelo, 5-7 Jan. 2005.

[130] C.J. Rodrigues. An alternative calculational proof of soundness of rely-garantee logic. Technical report, Department of Informatics, University of Minho, July 2007.

[131] C.J. Rodrigues. Transposing relations with adjunctions related to generic monads. Technical report, Department of Informatics, University of Minho, May 2008.

[132] J. Rutten. Universal coalgebra: A theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.

[133] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003.

[134] E. Schroder. Vorlesungen über die algebra der logik (exakte logic). *Dritter Band: Algebra und Logik der Relative. Teubner, Leipzig*, 1895.

[135] J. M. Spivey. *The Z Notation — A Reference Manual*. Series in Computer Science. Prentice-Hall International, 1992. 2nd ed.

[136] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, 1941.

[137] A. Tarski and S. Givant. A formalization of set theory without variables, 1987.

[138] B. von Karger and C. A. R. Hoare. Sequential calculus. *Information Processing Letters*, 53(3):123–130, 1995.

[139] J. von Wright. *A Lattice-theoretical Basis for Program Refinement*. PhD thesis, Abo Akademi University, September 1990.

[140] P. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, London, Sep. 1989. ACM.

[141] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[142] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

Appendix A

Algebra of Programming

This appendix presents some laws of the functional calculus and of the more general relational calculus.

A.1 Functional Calculus

- Composition

  Natural-id $f \cdot id = id \cdot f = f$

  Associativity $(f \cdot g) \cdot h = f \cdot (g \cdot h)$

- Product

  Universal $k = \langle f, g \rangle \Leftrightarrow \left\{ \begin{array}{l} \pi_1 \cdot k = f \\ \pi_2 \cdot k = g \end{array} \right.$

  Cancellation $\pi_1 \cdot \langle f, g \rangle = f$ and $\pi_2 \cdot \langle f, g \rangle = g$

  reflexion $\langle \pi_1, \pi_2 \rangle = id$

  fusion $\langle g, h \rangle \cdot f = \langle g \cdot f, h \cdot f \rangle$

  absorption $(i \times j) \cdot \langle g, h \rangle = \langle i \cdot g, j \cdot h \rangle$

  functor $(g \cdot h) \times (i \cdot j) = (g \times i) \cdot (h \times j)$

  functor-id $id_A \times id_B = id_{A \times B}$

- Coproduct

  Universal $k = [f, g] \Leftrightarrow \left\{ \begin{array}{l} k \cdot i_1 = f \\ k \cdot i_2 = g \end{array} \right.$

  Cancellation $[g, h] \cdot i_1 = g$ and $[g, h] \cdot i_2 = h$

  reflexion $[i_1, i_2] = id$

  fusion $f \cdot [g, h] = [f \cdot g, f \cdot h]$

  absorption $[g, h] \cdot (i + j) = [g \cdot i, h \cdot j]$

functor $(g \cdot h) + (i \cdot j) = (g + i) \cdot (h + j)$

functor-id $id_A + id_B = id_{A+B}$

- Exponentiation

Universal $k = \overline{f} \Leftrightarrow f = ap \cdot (k \times id)$

Cancellation $f = ap \cdot (\overline{f} \times id)$

Reflexion $\overline{ap} = id$

Fusion $\overline{g \cdot (f \times id)} = \overline{g} \times f$

Absorption $f^A \cdot \overline{g} = \overline{f \cdot g}$

Functor $(g \cdot h)^A = g^A \cdot h^A$

Functor-id $id^A = id$

A.2 Relational Calculus

- Axioms:$\cdot$,$\subseteq$

associativity $(R \cdot S) \cdot T = R \cdot (S \cdot T)$

identity $R = R \cdot id = id \cdot R$

$\subseteq$ **reflexivity** $R \subseteq R$

$\subseteq$ **transitivity** $R \subseteq S \ \wedge \ S \subseteq T \Rightarrow R \subseteq T$

$\subseteq$ **anti-simmetry** $R \subseteq S \ \wedge \ S \subseteq R \Rightarrow R = S$

monotonicity $S \subseteq T \ \wedge \ R \subseteq U \Rightarrow S \cdot R \subseteq T \cdot U$

- Axioms: $=$

ping-pong $R = S \equiv R \subseteq S \ \wedge \ S \subseteq R$

indirection $R = S \equiv \forall X.(X \subseteq R \equiv X \subseteq S)$

- Axioms:$\cap, \circ$

$\cap$ **universal** $X \subseteq (R \cap S) \equiv X \subseteq R \ \wedge \ X \subseteq S$

$\circ$ **involution** $R^{\circ\circ} = R$

$\circ$ **monotonicity** $R \subseteq S \equiv R^\circ \subseteq S^\circ$

$\circ$ **contravariance** $(R \cdot S)^\circ = S^\circ \cdot R^\circ$

left modular $(R \cdot S) \cap T \subseteq R \cdot (S \cap (R^\circ \cdot T))$

- Product

absorption $(R \times S) \cdot \langle X, Y \rangle = \langle R \cdot X, S \cdot Y \rangle$

cancellation $\langle R, S \rangle^\circ \cdot \langle X, Y \rangle = (R^\circ \cdot X) \cap (S^\circ \cdot Y)$

fusion $\langle R, S\rangle \cdot f = \langle R \cdot f, S \cdot f\rangle$

- Coproduct

cancellation $[R, S] \cdot [U, V] = (R \cdot U^\circ) \cup (S \cdot V^\circ)$

- **Shunting rules**

$f \cdot R \subseteq S \equiv R \subseteq f^\circ \cdot S$
$R \cdot f^\circ \subseteq S \equiv R \subseteq S \cdot f$

- **Equality**

$f \subseteq g \equiv f = g \equiv f \supseteq g$

- Dedekind Variations

left modular $(R \cdot S) \cap T \subseteq R \cdot (S \cap (R^\circ \cdot T))$
right modular $(R \cdot S) \cap T \subseteq (R \cap (T \cdot S^\circ)) \cdot S$
weak distr $(R \cdot S) \cap T \subseteq (R \cap (T \cdot S^\circ) \cdot (S \cap (R^\circ \cdot T))$

- $\cap$ Laws

$\cap$ **associativity** $(R \cap S) \cap T = R \cap (S \cap T)$
$\cap$ **commutativity** $R \cap S = S \cap R$
$\cap$ **idempotence** $R \cap R = R$
$\cap$ **abbreviation** $R \subseteq S \equiv R = R \cap S$
$\cap$ **cancellation** $R \cap S \subseteq R, R \cap S \subseteq S$
$\cap$ **left fusion** $T \cdot (R \cap S) \subseteq T \cdot R \cap T \cdot S$
$\cap$ **right fusion** $(R \cap S) \cdot T \subseteq R \cdot T \cap S \cdot T$

- $\circ$ Laws

reduction $R \subseteq R^\circ \equiv R = R^\circ$
wrap $R \subseteq R \cdot R^\circ \cdot R$
dist. over $\cap$ $(S \cap R)^\circ = S^\circ \cap R^\circ$

- $kernel$ Laws

definition $kerR = R^\circ \cdot R$
duality $kerR^\circ = imgR$
monotonicity $R \subseteq S \Rightarrow kerR \subseteq kerS$
symmetry $(kerR)^\circ = kerR$
intro $R \subseteq R \cdot kerR$

·-kernel $ker(R \cdot S) = S^\circ \cdot kerR \cdot S$

∩-kernel $R^\circ \cdot S \cap id \subseteq ker(R \cap S)$

- $image$ Laws

definition $imgR = R \cdot R^\circ$

duality $imgR^\circ = \ker R$

monotonicity $R \subseteq S \Rightarrow imgR \subseteq imgS$

symmetry $(imgR)^\circ = imgR$

intro $R \subseteq imgR \cdot R$

- Coreflexives

symm & transitive $R = R \cdot R^\circ = R \cdot R = R \cap id$

cancellation R coreflexive $\Rightarrow (R \cdot T \subseteq T) \wedge (T \cdot R \subseteq T)$

distributivity R coreflexive $\Rightarrow (R \cdot T) \cap S = R \cdot (T \cap S)$

- Order Taxonomy

reflexive $id_A \subseteq R$

coreflexive $R \subseteq id_A$

transitive $R \cdot R \subseteq R$

symmetric $R \subseteq R^\circ$

anti-symmetric $R \cap R^\circ \subseteq id_A$

connected $R \cup R^\circ = \top_A$

- Classification of Relations

entire $id \subseteq kerR$ (total relation)

simple $imgR \subseteq id$ (partial function)

surjection R° entire

injection R° simple

simple and surjective $imgR = id$

entire and injective $kerR = id$

Appendix B

Proofs

Proofs of a number of (auxiliary) results in the main body of this dissertation have been deferred to this appendix for economy of presentation. In this way, the flow of reading is not disturbed by the interleaving of such technical arguments. In this appendix we provide such proofs many of which illustrate the calculational style which we have adopted for agile reasoning based on the PF-transform of logical formulæ.

B.1 Transposability of simple relations

We want to prove the existence of function $\Gamma_{\mathsf{Id}+1}$ which converts *simple* relations into $(\mathsf{Id}+1)$-resultric functions and is such that $\Gamma_{\mathsf{Id}+1} = (\in_{\mathsf{Id}+1} \cdot)^{\circ}$, that is,

$$\in \cdot\, f = R \quad \equiv \quad f = \Gamma\, R$$

omitting the $\mathsf{Id}+1$ subscripts for improved readability. Our proof is inspired by [57]:

$$
\begin{array}{cl}
 & f = \Gamma\, R \\
\equiv & \quad \{ \text{ introduce } id \ \} \\
 & id \cdot f = \Gamma\, R \\
\equiv & \quad \{ \text{ coproduct reflexion } \} \\
 & [i_1, i_2] \cdot f = \Gamma\, R \\
\equiv & \quad \{ \text{ uniqueness of } 1 \xleftarrow{\ !\ } 1 \ = id \ \} \\
 & [i_1, i_2 \cdot !] \cdot f = \Gamma\, R \\
\equiv & \quad \{ \text{ require ``obvious'' properties (B.1,B.2) below} \} \\
 & [\Gamma\, id, \Gamma\, \bot] \cdot f = \Gamma\, R \\
\equiv & \quad \{ \text{ see (B.7) below } \} \\
 & (\Gamma[id, \bot]) \cdot f = \Gamma\, R \\
\equiv & \quad \{ \text{ the required fusion law stems from (B.6) below } \} \\
 & (\Gamma[id, \bot]) \cdot f = \Gamma\, R \\
\equiv & \quad \{ \ \Gamma \text{ is injective, see (B.4) below } \} \\
 & [id, \bot] \cdot f = R \\
\equiv & \quad \{ \text{ recall (3.5) } \} \\
 & \in \cdot f = R
\end{array}
$$

A number of facts were assumed above whose proof is on demand. Heading the list are

$$\Gamma\bot \ = \ i_2 \cdot ! \qquad \text{(B.1)}$$

$$\Gamma f \ = \ i_1 \cdot f \qquad \text{(B.2)}$$

which match our intuition about the introduction of "error" outputs: totally undefined relation $\bot$ should be mapped to the "everywhere-$Nothing$" function $i_2 \cdot !$, while any other simple relation R should "override" $i_2 \cdot !$ with the (non-$Nothing$) entries in $i_1 \cdot R$. Clearly, entireness of R will maximize the overriding — thus property (B.2).

Arrow $B+1 \xleftarrow{\Gamma\, R} A$ suggests its converse $B+1 \xrightarrow{(\Gamma R)^\circ} A$ expressed by

$$(\Gamma R)^\circ \ = \ [R^\circ, \cdots] \qquad \text{(B.3)}$$

which is consistent with (B.1) and (B.2) — it is easy to infer $(\Gamma\ \bot)^\circ = [\bot^\circ, !^\circ]$ and $(\Gamma\ f)^\circ = [f^\circ, \bot]$ from (2.25) — and is enough to prove that Γ has $\in = i_1^\circ$ as left-inverse,

$$\in \cdot\, \Gamma \ = \ id \qquad \text{(B.4)}$$

that is, that Γ is injective. We reason, for all R:

$$
\begin{array}{ll}
& i_1^\circ \cdot \Gamma R = R \\
\equiv & \quad \{ \text{ take converses } \} \\
& (\Gamma R)^\circ \cdot i_1 = R^\circ \\
\equiv & \quad \{ \text{ assumption (B.3) } \} \\
& [R^\circ, \cdots] \cdot i_1 = R^\circ \\
\equiv & \quad \{ \text{ coproduct cancellation } \} \\
& R^\circ = R^\circ
\end{array}
$$

The remaining assumptions in the proof require us to complete the construction of the transpose operator. Inspired by (B.1) and (B.2), we define

$$\Gamma_{\mathsf{Id}+1} R \stackrel{\text{def}}{=} (i_2 \cdot !) \dagger (i_1 \cdot R) \tag{B.5}$$

where $R \dagger S$, the "relation override" operator [1], is defined by $(R \cdot (id - \mathit{ker}\, S)) \cup S$, or simply by $R \dagger S = S \triangleleft S \triangleright R$ if we resort to relational *conditionals* [1]. This version of the override operator is useful in proving the particular instance of fusion (3.11) required in the proof: this stems from

$$(R \dagger S) \cdot f = (R \cdot f) \dagger (S \cdot f) \tag{B.6}$$

itself a consequence of a fusion property of the relational conditional [1].

It can be checked that (B.5) validates all other previous assumptions, namely (B.1,B.2) and (B.3). Because $R \dagger S$ preserves entirety on any argument and simplicity on both (simultaneously), $\Gamma\ R$ will be a function provided R is simple.

The remaining assumption in the proof stems from equalities

$$[\Gamma\ id, \Gamma\ \bot] = \Gamma[id, \bot] = \Gamma(i_1^\circ) = \mathit{img}\, i_1 \cup \mathit{img}\, i_2 = id \tag{B.7}$$

which arise from (B.5) and the fact that i_1 and i_2 are (dis)jointly surjective injections.

B.2 Proof of (4.6)

First note that

$$
\begin{array}{lll}
\Xi_h t & = & \rho\,(\in \cdot\, (t \mathbin{\dot\cap} \Lambda(h^\circ))) \\
& = & \quad \{ \text{ since } \in \cdot\, (f \mathbin{\dot\cap} g) = (\in \cdot f) \cap (\in \cdot g) \ \} \\
& & \rho\,((\in \cdot\, t) \cap (\in \cdot\, \Lambda h^\circ)) \\
& = & \quad \{ \text{ cancellation } \} \\
& & \rho\,(\in \cdot\, t \cap h^\circ)
\end{array}
\tag{B.8}
$$

Thus (4.6) rewrites to

$$
\begin{array}{ll}
 & (4.6) \\
= & \quad \{\ \text{by (4.5) and (B.8)}\ \} \\
 & \in \cdot\, \Theta_h(\rho\,(\in \cdot\, t \cap h^\circ)) \quad \subseteq \quad \in \cdot\, t \\
\equiv & \quad \{\ \text{definition of } \Theta_h \text{ and cancellation} \} \\
 & (\rho\,(\in \cdot\, t \cap h^\circ)) \cdot h^\circ \quad \subseteq \quad \in \cdot\, t \\
\equiv & \quad \{\ \text{shunting}\ \} \\
 & \rho\,(\in \cdot\, t \cap h^\circ) \quad \subseteq \quad \in \cdot\, t \cdot h \\
\Leftarrow & \quad \{\ \rho\, R \subseteq \mathit{img}\ R \text{ and } R \cap S \subseteq S\ \} \\
 & \mathit{img}\,(\in \cdot\, t \cap h^\circ) \quad \subseteq \quad \in \cdot\, t \cdot h \\
\Leftarrow & \quad \{\ \text{definition of } \mathit{img} \text{ and converses}\ \} \\
 & (\in \cdot\, t \cap h^\circ) \cdot (h \cap (\in \cdot\, t)^\circ) \quad \subseteq \quad \in \cdot\, t \cdot h \\
\Leftarrow & \quad \{\ \text{by} \ \ T \cdot (R \cap S) \ \subseteq \ (T \cdot R) \cap (T \cdot S) \ \text{ and } \ (R \cap S) \cdot T \ \subseteq \ (R \cdot T) \cap (S \cdot T) \ \ \} \\
 & \in \cdot\, t \cdot h \cap \mathit{img}\,(\in \cdot\, t) \cap \mathit{ker}\, h \cap (\in \cdot\, t \cdot h)^\circ \quad \subseteq \quad \in \cdot\, t \cdot h \\
\Leftarrow & \quad \{\ R \cap S \subseteq S\ \} \\
 & \textsc{True}
\end{array}
$$

B.3 Proof of (4.7)

$$
\begin{array}{ll}
& \Xi_h(\Theta_h S) = S \\
\equiv & \qquad \{ \text{ definitions and (B.8) } \} \\
& \rho\,(\in \cdot\, (\Theta_h S) \cap h^\circ) = S \\
\equiv & \qquad \{ \text{ definition } \} \\
& \rho\,(\in \cdot\, (\Lambda(S \cdot h^\circ)) \cap h^\circ) = S \\
\equiv & \qquad \{ \text{ cancellation } \} \\
& \rho\,(S \cdot h^\circ \cap h^\circ) = S \\
\equiv & \qquad \{ \ S \cdot h^\circ \subseteq h^\circ \text{ since } S \text{ is coreflexive (see (B.9) below)} \} \\
& \rho\,(S \cdot h^\circ) = S \\
\equiv & \qquad \{ \text{ range of composition } \} \\
& \rho\,(S \cdot \rho\, h^\circ) = S \\
\equiv & \qquad \{ \ h \text{ is entire } \} \\
& \rho\, S = S \\
\equiv & \qquad \{ \ S \text{ is coreflexive } \} \\
& S = S
\end{array}
$$

Auxiliary result

$$
S \cdot R \subseteq R \quad \Leftarrow \quad S \text{ is coreflexive} \tag{B.9}
$$

is immediate:

$$
\begin{array}{ll}
& S \cdot R \subseteq R \\
\Leftarrow & \qquad \{ \text{ monotonicity } \} \\
& S \subseteq id \ \wedge \ R \subseteq R \\
\equiv & \qquad \{ \ S \text{ is coreflexive } \} \\
& \textsc{True}
\end{array}
$$

B.4 Split-fusion of simple relations

Function q in fusion property (3.28) generalizes to any simple relation M,

$$
\langle R, S \rangle \cdot M \quad = \quad \langle R \cdot M, S \cdot M \rangle \tag{B.10}
$$

cf.

$$
\begin{array}{ll}
& \langle R, S\rangle \cdot M \\
= & \quad \{ \text{ definition of split} \} \\
& ((\pi_1^\circ \cdot R) \cap (\pi_2^\circ \cdot S)) \cdot M \\
= & \quad \{ \text{ since } M \text{ is simple} \} \\
& (\pi_1^\circ \cdot R \cdot M) \cap (\pi_2^\circ \cdot S \cdot M) \\
= & \quad \{ \text{ definition of split} \} \\
& \langle R \cdot M, S \cdot M\rangle
\end{array}
$$

B.5 Nested Join

In this section we prove a number of results related to the nested join operator (4.14) studied in chapter 4. We begin with the proof of (4.26):

$$
\begin{array}{ll}
& \langle R, S\rangle \\
= & \quad \{ \text{ reflexion} \} \\
& \langle \pi_1, \pi_2\rangle \cdot \langle R, S\rangle \\
= & \quad \{ \text{ fusion for simple relations (B.10) } \} \\
& \langle \pi_1 \cdot \langle R, S\rangle, \pi_2 \cdot \langle R, S\rangle\rangle \\
= & \quad \{ \text{ projections applied to splits (4.24)} \} \\
& \langle R \cdot \delta\, S, S \cdot \delta\, R\rangle
\end{array}
$$

The following result, valid for arbitrary function f and relation R, will be needed in the sequel:

Lemma B.1

$$
f \cdot \delta\, R = R \quad \Leftarrow \quad R \subseteq f \tag{B.11}
$$

Proof: *Case $R \subseteq f \cdot \delta\, R$ is immediate by monotonicity of $(\ \cdot \delta\, R)$. We are left with:*

$$
\begin{array}{ll}
& f \cdot \delta\, R \subseteq R \\
\equiv & \quad \{ \textit{ shunting rules (2.15,5.31) } \} \\
& R^\circ \subseteq f^\circ \\
\equiv & \quad \{ \textit{ converses } \} \\
& R \subseteq f
\end{array}
$$

Fact (4.27) is an instance of the following lemma:

Lemma B.2 *Adjoint* usc *(4.20) satisfies the following property, for simple* X*:*

$$\overline{usc\ X} \cdot \delta\, X = X \tag{B.12}$$

The proof is based on the anti-symmetry of $\dot{\subseteq}$ *(4.21):*

$$S = T \equiv S \mathrel{\dot{\subseteq}} T \wedge T \mathrel{\dot{\subseteq}} S \tag{B.13}$$

Proof: *First, we deal with case* $X \mathrel{\dot{\subseteq}} \overline{usc\ X} \cdot \delta\, X$*:*

$$\begin{array}{cl}
 & X \mathrel{\dot{\subseteq}} \overline{usc\ X} \cdot \delta\, X \\
\equiv & \qquad \{\ \textit{(4.21)}\ \} \\
 & X \subseteq (\subseteq) \cdot \overline{usc\ X} \cdot \delta\, X \\
\equiv & \qquad \{\ \textit{shunting}\ \} \\
 & X \cdot \delta\, X \subseteq (\subseteq) \cdot \overline{usc\ X} \\
\equiv & \qquad \{\ \textit{trivia}\ \} \\
 & X \mathrel{\dot{\subseteq}} \overline{usc\ X} \\
\equiv & \qquad \{\ \textit{GC (4.20)}\} \\
 & \textsc{True}
\end{array}$$

The other case is:

$$\begin{array}{cl}
 & \overline{usc\ X} \cdot \delta\, X \\
= & \qquad \{\ \textit{quantifier calculus and } (=) \textit{ is an upper adjoint}\} \\
 & \langle \bigcap\ S\ :\ X \subseteq \overline{S} :\ \overline{S} \rangle \cdot \delta\, X \\
\subseteq & \qquad \{\ \textit{meet right fusion}\} \\
 & \langle \bigcap\ S\ :\ X \subseteq \overline{S} :\ \overline{S} \cdot \delta\, X \rangle \\
= & \qquad \{\ \textit{(B.11}\} \\
 & \langle \bigcap\ S\ :\ X \subseteq \overline{S} :\ X \rangle \\
= & \qquad \{\ \textit{idempotency of meet}\} \\
 & X
\end{array}$$

B.6 Equating Simple Relations

Lemma B.3 (Equality of simple relations) *For simple* S, R*,*

$$R = S \equiv \quad R \subseteq S\ \wedge\ \delta\, S \subseteq \delta\, R \tag{B.14}$$

Proof:

$$
\begin{array}{cl}
& R = S \\
\equiv & \quad \{ \text{ \textit{anti-symmetry; Leibniz} } \} \\
& R \subseteq S \ \wedge \ S \subseteq R \ \wedge \ \delta\, S = \delta\, R \\
\equiv & \quad \{ \text{ \textit{shunt on simple S (5.30)} } \} \\
& R \subseteq S \ \wedge \ \delta\, S \subseteq S^{\circ} \cdot R \ \wedge \ \delta\, S = \delta\, R \\
\equiv & \quad \{ \text{ \textit{substitution of} } \delta\, S \text{ \textit{by} } \delta\, R \ \} \\
& R \subseteq S \ \wedge \ \delta\, R \subseteq S^{\circ} \cdot R \ \wedge \ \delta\, S = \delta\, R \\
\equiv & \quad \{ \text{ \textit{shunt on simple R (5.31)} } \} \\
& R \subseteq S \ \wedge \ R^{\circ} \subseteq S^{\circ} \ \wedge \ \delta\, S = \delta\, R \\
\equiv & \quad \{ \text{ \textit{converses ; anti-symmetry} } \} \\
& R \subseteq S \ \wedge \ R \subseteq S \ \wedge \ \delta\, S \subseteq \delta\, R \ \wedge \ \delta\, R \subseteq \delta\, S \\
\equiv & \quad \{ \text{ \textit{last conjunct implied by first} } \} \\
& R \subseteq S \ \wedge \ \delta\, S \subseteq \delta\, R
\end{array}
$$

□

B.7 Mathematical Background for Chapter 5

The following conditional equality

$$(S \cap T) \cdot R = (S \cdot R) \cap (T \cdot R) \quad \Leftarrow \quad T \cdot \mathit{img}\, R \subseteq T \vee S \cdot \mathit{img}\, R \subseteq S \tag{B.15}$$

and its converse-dual,

$$R \cdot (S \cap T) = (R \cdot S) \cap (R \cdot T) \quad \Leftarrow \quad (\mathit{ker}\, R) \cdot T \subseteq T \vee (\mathit{ker}\, R) \cdot S \subseteq S \tag{B.16}$$

are laws (11.20) and (11.17) of [12], respectively. Exercise Ex.11.22 in [12] addresses law

$$\langle \forall\, S, T \ :: \ R \cdot S \cap T = R \cdot (S \cap T) \rangle \quad \equiv \quad R \subseteq id \tag{B.17}$$

whose converse-dual is:

$$\langle \forall\, S, T \ :: \ S \cdot R \cap T = (S \cap T) \cdot R \rangle \quad \equiv \quad R \subseteq id \tag{B.18}$$

The proof of (5.23) assumes conditional equality

$$\delta\, S = \delta\, (S \cap R) \quad \Leftarrow \quad S \vdash R \tag{B.19}$$

which is easy to calculate:

$$
\begin{array}{ll}
& S \vdash R \\
\equiv & \qquad \{ \text{ definition } \} \\
& R \cdot \delta\, S \subseteq S \;\wedge\; \delta\, S \subseteq \delta\, R \\
\equiv & \qquad \{ \text{ since } R \cdot \delta\, S \subseteq R \text{ ; universal property of } \cap \text{ ; coreflexives } \} \\
& R \cdot \delta\, S \subseteq S \cap R \;\wedge\; \delta\, S \cdot \delta\, R = \delta\, S \\
\Rightarrow & \qquad \{ \; \delta \text{ is monotonic ; } \delta \text{ of composition } \} \\
& \delta\,(\delta\, R \cdot \delta\, S) \subseteq \delta\,(S \cap R) \\
\equiv & \qquad \{ \text{ substitution; } \delta \text{ is idempotent } \} \\
& \delta\, S \subseteq \delta\,(S \cap R) \\
\equiv & \qquad \{ \text{ since } \delta\,(S \cap R) \subseteq \delta\, S \text{ by monotonicity of } \delta \text{ } \} \\
& \delta\, S = \delta\,(S \cap R)
\end{array}
$$

□

The following are two useful facts:

$$
g \cdot \mathit{ker}\,(R \cdot g) \subseteq (\mathit{ker}\, R) \cdot g \tag{B.20}
$$

$$
f \cdot (\mathit{ker}\, g) \subseteq (\mathit{ker}\, g) \cdot f \quad \Leftarrow \quad f \cdot g = g \cdot f \tag{B.21}
$$

Proof of (B.20):

$$
\begin{array}{ll}
& g \cdot \mathit{ker}\,(R \cdot g) \\
= & \qquad \{ \text{ expand kernel (2.1) } \} \\
& g \cdot g^{\circ} \cdot R^{\circ} \cdot R \cdot g \\
= & \qquad \{ \text{ introduce image (2.2) and kernel } \} \\
& (\mathit{img}\, g) \cdot (\mathit{ker}\, R) \cdot g \\
\subseteq & \qquad \{ \; \mathit{img}\, g \text{ is coreflexive } \} \\
& (\mathit{ker}\, R) \cdot g
\end{array}
$$

Proof of (B.21):

$$
\begin{array}{ll}
& f \cdot (\mathit{ker}\, g) \subseteq (\mathit{ker}\, g) \cdot f \\
\equiv & \qquad \{ \text{ expand } \mathit{ker}\, g \text{ and shunt as much as possible } \} \\
& \mathit{ker}\, g \subseteq f^\circ \cdot (\mathit{ker}\, g) \cdot f \\
\equiv & \qquad \{ \text{ converse of composition } \} \\
& \mathit{ker}\, g \subseteq \mathit{ker}\, (g \cdot f) \\
\equiv & \qquad \{ \text{ since } f \cdot g = g \cdot f \ \} \\
& \mathit{ker}\, g \subseteq \mathit{ker}\, (f \cdot g) \\
\equiv & \qquad \{ \ \mathit{ker}\, f \text{ is reflexive } \} \\
& \textsc{True}
\end{array}
$$

Concerning (5.34), we can use formula (5.16) for $\vdash_{post}$ and reason at pointfree level:

$$
\begin{array}{ll}
& f \cdot \vdash_{post} \\
= & \qquad \{ \ (5.16) \ \} \\
& f \cdot (\subseteq^\circ \cap (\mathit{ker}\, \delta)) \\
\subseteq & \qquad \{ \text{ composition is monotonic } \} \\
& (f \cdot \subseteq^\circ) \cap (f \cdot \mathit{ker}\, \delta) \\
\subseteq & \qquad \{ \ f \text{ is } \subseteq\text{-monotonic by hypothesis and (B.21) for } f, g := \mathsf{F}, \delta \ \} \\
& (\subseteq^\circ \cdot f) \cap (\mathit{ker}\, \delta \cdot f) \\
= & \qquad \{ \text{ (B.15) — since } f \text{ is simple} \} \\
& (\subseteq^\circ \cap (\mathit{ker}\, \delta)) \cdot f \\
= & \qquad \{ \ (5.16) \ \} \\
& \vdash_{post} \cdot f
\end{array}
\tag{B.22}
$$

Since every relator is $\subseteq$-monotonic and commutes with δ (domain), (5.35) is just a corollary of (5.34). Concerning (5.36), we start by introducing variables,

$$
\begin{array}{ll}
& \mathsf{F} \cdot \vdash_{pre} \ \subseteq \ \vdash_{pre} \cdot \mathsf{F} \\
\equiv & \qquad \{ \text{ shunting } \} \\
& \vdash_{pre} \ \subseteq \ \mathsf{F}^\circ \cdot \vdash_{pre} \cdot \mathsf{F} \\
\equiv & \qquad \{ \text{ rule (2.6) } \} \\
& S \vdash_{pre} R \ \Rightarrow \ \mathsf{F}\, S \vdash_{pre} \mathsf{F}\, R
\end{array}
$$

and proceed:

$$
\begin{array}{cl}
& \mathsf{F}\,S \vdash_{pre} \mathsf{F}\,R \\
\equiv & \quad \{ \ (5.14) \ \} \\
& \mathsf{F}\,R \cdot \delta\,(\mathsf{F}\,S) = \mathsf{F}\,S \\
\equiv & \quad \{ \ \mathsf{F} \text{ commutes with } \delta \} \\
& \mathsf{F}\,R \cdot \mathsf{F}(\delta\,S) = \mathsf{F}\,S \\
\equiv & \quad \{ \ \mathsf{F} \text{ commutes with composition } \} \\
& \mathsf{F}(R \cdot \delta\,S) = \mathsf{F}\,S \\
\Leftarrow & \quad \{ \ \text{Leibniz} \ \} \\
& R \cdot \delta\,S = S \\
\equiv & \quad \{ \ (5.14) \ \} \\
& S \vdash_{pre} R
\end{array}
$$

□

B.8 Guards and Predicate Semantics

Let p be a predicate on type A. Univocally associated with p we find:

- coreflexive relation $A \xleftarrow{[\![p]\!]} A$ defined by

$$b\ [\![p]\!]\ a \equiv (b = a) \wedge (p\ a) \tag{B.23}$$

- guard $A + A \xleftarrow{p?} A$ defined pointfree as $(p?)^{\circ} = [[\![p]\!], [\![\neg p]\!]]$, cf. picture

$$
\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "{[\![p]\!]}"'] & A + A & A \arrow[l, "i_2"'] \arrow[dl, "{[\![\neg p]\!]}"] \\
& A \arrow[u, "p?"] &
\end{tikzcd}
$$

We have the following lemma:

Lemma B.4 (Relationship between $p?$ and $[\![p]\!]$) *Instead of the existent predicate coreflexive semantics,*

$$(p?)a = \begin{cases} p\ a & \Rightarrow\ i_1\ a \\ \neg\,(p\ a) & \Rightarrow\ i_2\ a \end{cases}$$

we may define a new predicate coreflexive semantics in the pointfree style using guards:

$$\llbracket p \rrbracket = (p?)^\circ \cdot i_1 \tag{B.24}$$

$$\llbracket \neg p \rrbracket = (id - \llbracket p \rrbracket) = (p?)^\circ \cdot i_2 \tag{B.25}$$

Proof: *Immediate by coproduct cancellation.*